

SPECIAL INSTRUCTIONS FOR 3PC

Due to an oversight, the TEXT-MASTER program indicated in the enclosed Overview was inadvertently omitted from the enclosed Program Disk. Rather than delay this module, we will include TEXT-MASTER on your Module 4 Program Disk. We apologize for any inconvenience this may cause.

SI 924(802)

INTRODUCING MODULE 3:
HOW TO DESIGN THE SOLUTION
AND ARRANGE IT LOGICALLY

In your previous module you had some opportunities to work on problem solving. You learned how to formulate a basic problem statement and you experimented with rearranging this statement into new configurations.

You are now ready for the next step! In Module 3, you will complete the initial design phase process. You will learn about flowcharts and Chapin diagrams, two design techniques that are essential program structuring tools.

The interactive diskette will provide additional details on your checkbook register and phone book programs (which you will complete in Module 4). It also contains the program subroutines for generating the blank video display you worked with in the preceeding module.

At the conclusion of this module, you will be able to complete the initial paper design of a program. You'll then be prepared to enter the realm of coding.

Module 4 will focus on high-level languages, including BASIC, Fortran, and COBOL. You'll acquire the skills to begin running programs of your own at this stage.

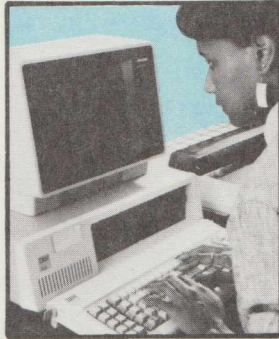
We're certain you are going to enjoy learning more about designing and arranging solutions. Module 3 awaits you.

Sincerely,



Kenneth J. Bigelow
Project Engineer

CONTEMPORARY PROGRAMMING AND SOFTWARE DESIGN SERIES



THIS IS Module 3 in McGraw-Hill's Contemporary Programming and Software Design Series. In this Module, you will complete your study of the initial design phase of the programming process.

The disk provided with this module is intended for COMMODORE 64 computers and for C-128 computers operating in C-64 mode. Unless otherwise specified, all programs on this disk are written in BASIC.

Although it is not strictly necessary, we strongly urge that you place a write-protect tab on the enclosed disk and then use BACKUP or a similar backup utility program to make a copy of this disk. Use the backup copy when working with this module. Then, if your working disk should accidentally become damaged or erased, you can make a new backup and continue your studies.

As you progress through this module, you will find that some of the files on the enclosed Program Disk are programs saved in ASCII format. These are usually partial programs that you will merge with existing programs you already have. Commodore BASIC does not normally permit this, but it can be "fooled" into interpreting such a program from disk instead of from the keyboard. In this module you will see how you can save such programs and routines to disk and how you can load them back into your computer. You will find these techniques useful in developing and updating programs of all kinds.

To begin this module, *turn on your disk drive (and printer, if you have one connected), and then your computer. If you are using a C-128 computer, type GO64 and press <return>. If necessary, type*

Y to confirm this command. Then, place your Program Disk or its backup into Drive #8 and close the drive door. Type in the commands:

*LOAD "WELCOME",8<return>
RUN<return>*

When your computer directs you to turn to your Learning Guide, begin with the Introduction.

Contemporary Programming & Software Design

**FOR THE COMMODORE 64
AND 128 COMPUTERS**

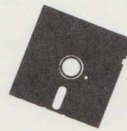


TABLE OF CONTENTS

**HOW TO DESIGN THE SOLUTION
AND ARRANGE IT LOGICALLY**

Introduction	1
Track 1: The Solution Design Procedure	3
Track 2: Detailing the Solution Procedure	11
Track 3: Drawing Flow Charts.	19
Track 4: The Chapin Diagram	33
Track 5: Corrections and Adjustments	43
Track 6: Some Practical Experience with the Computer	53

TABLE OF CONTENTS

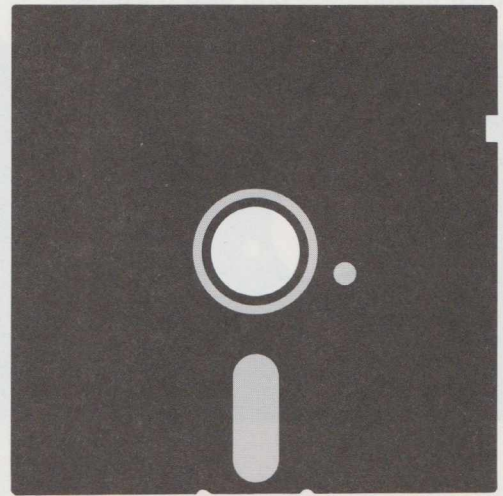
McGraw-Hill Continuing Education Center
3939 Wisconsin Avenue
Washington, D.C. 20016

Copyright 1987 by McGraw-Hill, Inc. All rights reserved. Manufactured in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this package may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

0-07-580066-7

Contemporary Programming & Software Design

MODULE 3



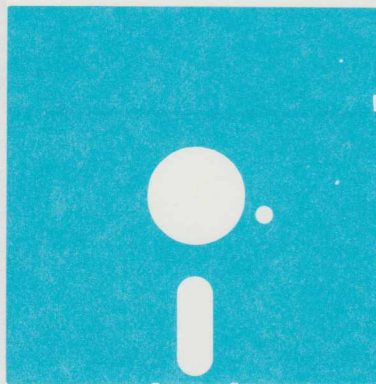
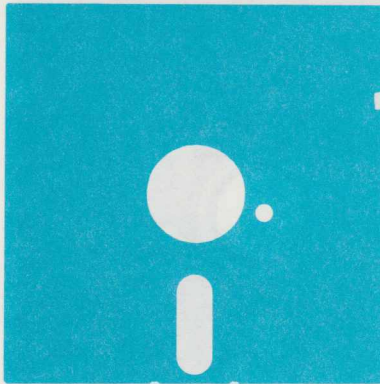
HOW TO DESIGN THE SOLUTION AND ARRANGE IT LOGICALLY

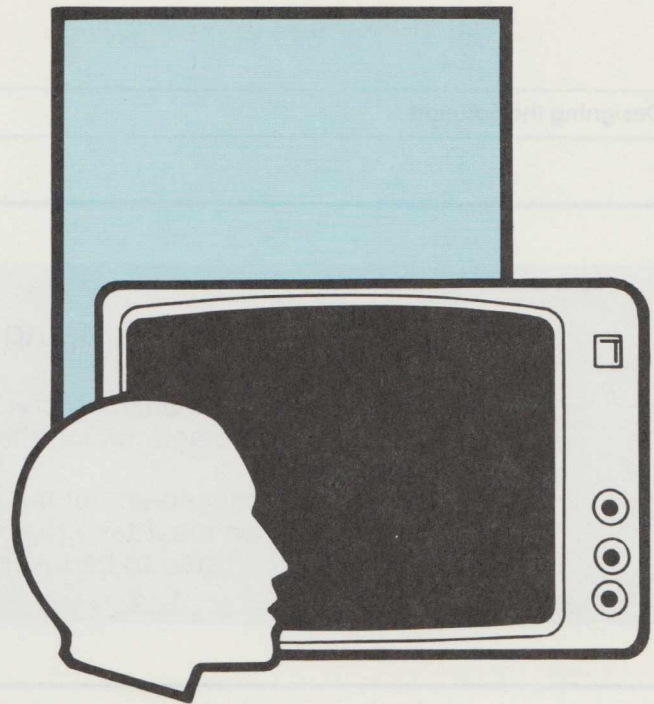
In the last module, we described the general procedure that we wanted the computer to follow when solving a problem. We set up some examples of practical programs and we saw the correlation between the procedure outline and the actual program. However, in real programming, there is a step in between that we skipped over. This is the step that we will explore in this module.

The intermediate step that we bypassed involves the generation of a detailed "map" or chart that precisely defines the path that the computer will take to solve its problem. Without this map or chart, writing a working program directly from the solution outline and verbal descriptions requires a substantial amount of experience and usually several back-and-forth sessions of "try it again and correct it again." As you gain experience in writing your own programs, you may wish to skip the mapping step, at least for small tasks. However, for larger tasks and for programmers with less experience, mapping is highly desirable.

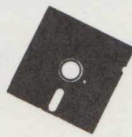
There are several different methods used to map out the solution to a problem. Perhaps the best known method is the flow chart. Each technique was developed to provide certain advantages over other mapping techniques, or to simplify the mapping of certain types of programs. Each method has both advantages and disadvantages with respect to other methods.

In this module, we will explore two common mapping techniques: the flow chart and the Chapin diagram. We will compare them and explore their advantages and disadvantages.





TRACK 1



THE SOLUTION DESIGN PROCEDURE

As you already know, we have broken up the general programming process into seven phases, as shown in [Figure 1-1](#). Of course, each phase leads logically and directly to the next, to complete the overall programming process. However each phase of the programming process has its own requirements and its own logical procedures.

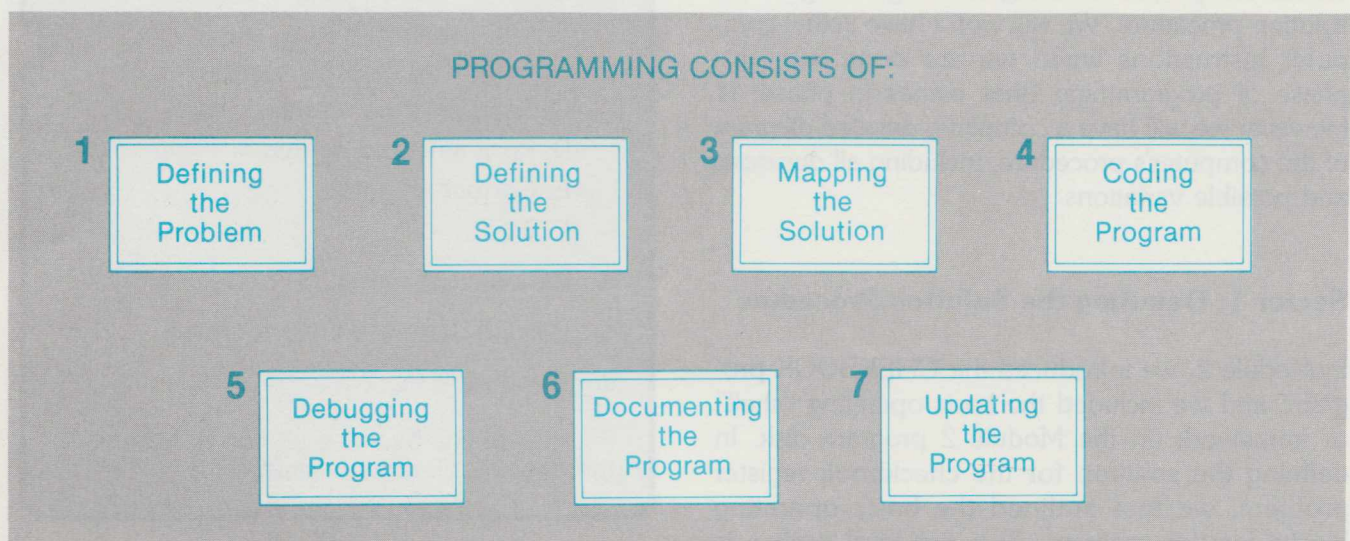


Figure 1-1. The seven basic phases of programming.

3. Mapping the Solution

Mapping the solution consists of detailing, organizing, and diagramming the computer's procedure.

Earlier we produced an outline of the steps to be followed by the computer. Now, we must take that outline and expand it to include every step to be taken by the computer.

Figure 1-2. Mapping the solution.

We have already covered the definition phases of programming (boxes 1 and 2 in [Figure 1-1](#)) in the previous module. Now we will see how to convert the solution definition from phase 2 into a fully detailed computer procedure. This is box number 3, mapping the solution. In another module, we will take the fully mapped procedure and use it to perform phase 4, where we will actually code the program and enter it into the computer.

Refer to [Figure 1-2](#). Mapping the solution means more than just drawing a picture such as a flow chart or Chapin diagram. (We will see exactly what these are a little later in this module.) It also includes the precise detailing and organizing of the solution procedure. We still won't have actual computer instructions when we are done with this phase of programming (that comes in phase 4). However we will have a complete, detailed diagram of the computer's procedure, including all decisions and possible variations.

Sector 1: Detailing the Solution Procedure

In Module 2, we introduced the CHEKBOOK program, and we included the basic operating "shell" or framework on the Module 2 program disk. In defining the solution for the checkbook register problem, we first outlined the basic operating blocks for the program. This first-level outline is shown in [Figure 1-3](#).

Of course, this outline just provides the general sequence of events for this program; it does not describe the actual operations needed to perform any of the tasks listed. The same holds true for the individual routine outlines we saw in Module 2. These outlines define the basic solution procedure by identifying what actions the computer needs to take. The next step is to add the details of computer operation to each step.

1. Get today's date.
2. List available program options.
 - A. Enter new checks.
 - B. Reconcile monthly statement.
 - C. Scan existing entries.
 - D. Print selected entries.
 - E. Correct an entry.
 - F. Exit.
3. Request user input.
4. Execute user-specified routine.
5. Return to Step 2.

Figure 1-3. The basic sequence of events for the checkbook register program.

As an example, consider Step 1 from **Figure 1-3**: getting the current date. Let's list the detailed sequence of operations that the computer must go through to get the date. Remember that while the Commodore computers do not maintain the date internally, some other computers and versions of BASIC do. Therefore, we will allow for both possibilities here:

1. If BASIC does not include a DATE\$ function, go directly to Step 6.
2. Display current value of DATE\$ with explanatory message.
3. Ask if displayed date is correct.
4. If date is correct, skip to Step 9.
5. If date is not correct, or if DATE\$ is not a function in your version of BASIC, continue with Step 6.
6. Request current date in MM/DD/YY format.
7. If BASIC includes DATE\$ function, reset DATE\$ to the new date. (You cannot INPUT data to the DATE\$ function. You must INPUT to a variable and then use the variable to reset DATE\$.)
8. If BASIC does not include the DATE\$ function, INPUT the current date to a variable named DATE\$ or DA\$.
9. Continue with the next step in the program.

This list includes the possible choices and variations that the programmer may have to contend

with. In this case, there are two basic decisions to make: whether or not BASIC includes the DATE\$ function, and whether or not the date currently held in the computer is correct. The first decision will be made by the programmer at the time of coding the program for the specific computer and language. Or the programmer can write the program in such a way as to enable it to handle either possibility. The second decision will be made by the user, while the program is running.

Sector 2: Charting the Detailed Procedure

Now that we have a detailed procedure for obtaining the current date, we can develop a chart, or map, of the computer procedure. One of the most common ways of doing this is to use a *flow chart*,



such as the one shown in Figure 1-4. This flow chart is basically a pictorial or graphic representation of the list of steps in Sector 1. The shape of each box in the chart is determined by the kind of function described in that box. The arrows show the flow of the program logic from step to step. Each box may indicate one or several steps, according to the requirements of the steps. However each step must be explicitly listed within its box.

The ellipses at the top and the bottom of the flow chart indicate terminal points in the sequence. That is, these are the entry and exit points. The top ellipse is always the entry point, and may indicate not only that fact, but also the way the overall program will reach this particular routine. The bottom ellipse is always the exit point from this routine, and may also indicate the routine that will be next in the logical sequence, if such an indication is appropriate.

The diamond-shaped boxes indicate decisions. These are points where the program can cause the computer to skip part of the program or else repeat part of the program, according to the current circumstances or situation. The question is asked or the desired comparison indicated inside the box, and all possible answers or results are labeled on the arrows that indicate the program flow in case of that result. Most decisions are listed as yes-no questions having two possible answers. This is the case in Figure 1-4. Comparisons generally have two or three possible outcomes, and normally involve the comparison of two numbers or character codes. In such cases, there may be the notation "A:B" inside

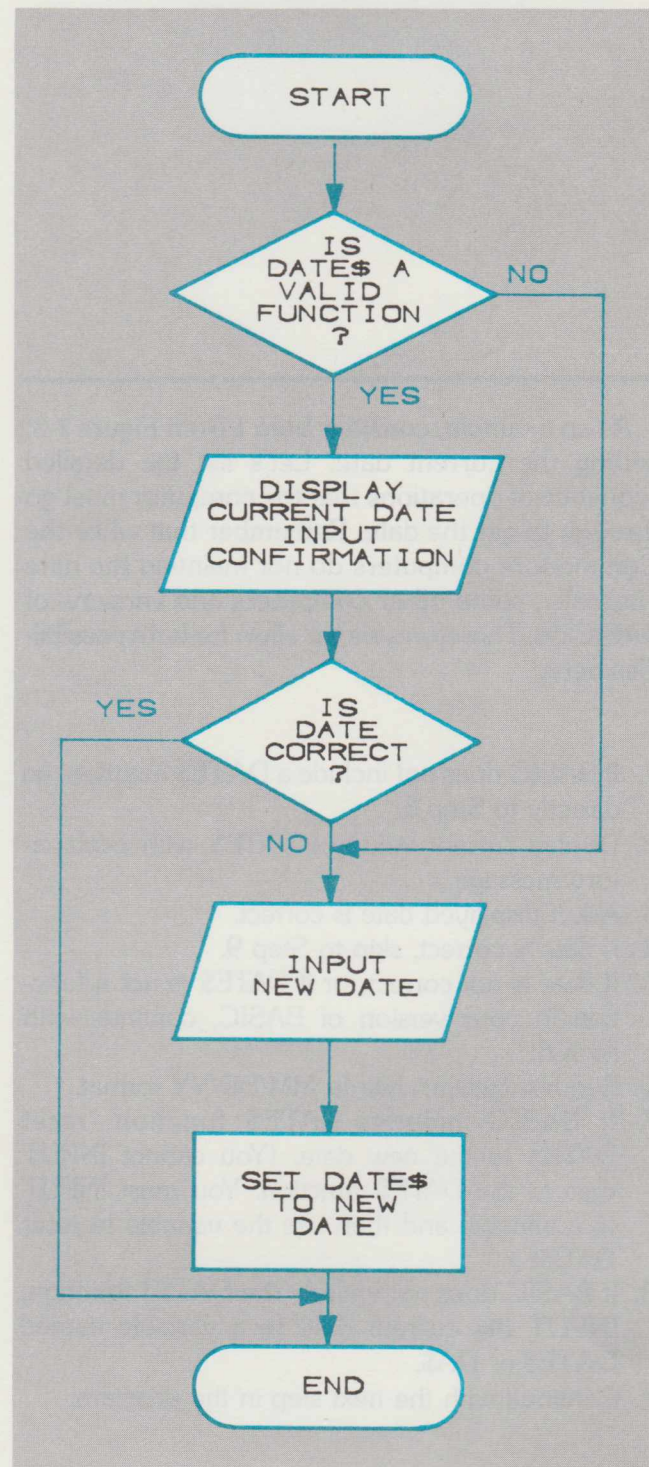


Figure 1-4. A flow chart of the procedure to obtain the current date.

the box to indicate a comparison between A and B. Then the labels “ $A < B$ ”, “ $A = B$ ”, and “ $A > B$ ” will appear on either two or three outgoing arrows to denote program flow in the event that A is less than, equal to, or greater than B, respectively.

The parallelograms (the boxes with horizontal top and bottom lines and slanting sides) define input and output operations. These are operations to display something to the screen, send it to the printer, or request user input from the keyboard. A group of I/O operations may be included in one box.

The rectangle indicates any internal process. It may indicate one or more calculations or memory operations which may be carried out in sequence, without contact with the outside world.

Flow charts are drawn so that the general flow of the program is from top to bottom. A complex flow chart may show program flow from left to right as well as from top to bottom. Such charts are arranged as several columns of symbols placed side by side, and the program flow is still from top to bottom, moving from one column to the next.

In addition, decisions are so stated that the most complete program flow is in line, from top to bottom. Choices that bypass parts of the program are shown that way. This practice leaves the program steps aligned with each other, as they are in Figure 1-4. Note in the figure that the two choices that skip over parts of the sequence go off to one side or the other, and then back to the main program sequence.

The flow chart of Figure 1-4 doesn't really contain any more detail or information than the list in Sector 1. It just presents the information in a different fashion. This pictorial display is often more useful, however, because it can help to more clearly define the exact sequence of operations, especially in regard to the different choices that may arise from decisions. Also, the flow chart can be helpful in showing where additional detail should be added to provide a complete definition of program behavior.

The flow chart also contains only seven boxes of various shapes, where our list of steps in Sector 1 had nine steps in it. Generally this will be the case for two reasons: first, you can put more than one step in a single box; and second, our list included two steps to denote alternate decisions, where the flow chart shows a single box for the entire decision.

Sector 3: Another Method of Charting the Solution

The flow chart is a useful tool for visualizing the flow pattern of a program, and for making sure that all of the required detail is present. However it does have one drawback—it uses up a lot of space on the paper. Very often, this means that even a single module or option routine within a program can

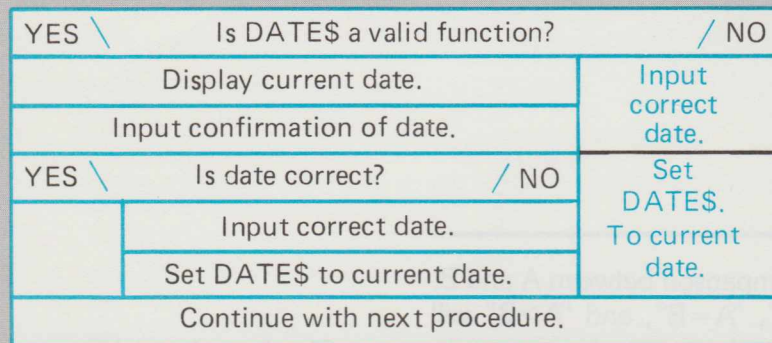


Figure 1-5. A Chapin diagram of the procedure to get the current date.

take up several pages of flow chart. Thus, the price of the pictorial approach with its increased legibility is the amount of space required to hold it.

Fortunately, there are other ways of diagramming the program sequence. In fact, some were developed specifically to retain some of the pictorial approach and still use the page efficiently. [Figure 1-5](#) shows such a diagram. This one is known as a *Chapin diagram*.

As you can see, the Chapin diagram is much more compact than the flow chart. Each step or operation uses only one line on the page. Furthermore, no arrows are required on the chart. Decisions are indicated normally by the words YES and NO as shown, and operations based on a given decision are listed under the appropriate decision response. If different actions are required depending on the decision choice, they are listed side by side, with a vertical line between them to show that these are completely separate flow paths.

The Chapin diagram for the date routine occupies only seven lines on the page, rather than seven multiline boxes along with extra lines and arrows. It still allows for choices and decisions. Most program steps can be defined on a single line. In addition, this sort of diagram lends itself very

well to the modular programming approach. That is, a single page will normally hold the complete description of a program module. If it does not, chances are that your module is too complicated, and should be split into two or more smaller modules. The resulting program is likely to be well structured and easy to follow.

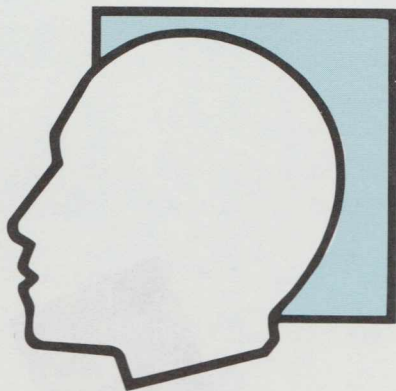
The Chapin diagram does have a few other drawbacks. For one thing, it only supports two-way decisions. That is, it supports decisions whereby the program must choose between two alternatives. A choice among three or more alternatives is more difficult to show.

Of course, we can show a multiple decision as a sequence of two-way decisions, with each NO answer leading to another decision. This produces the second drawback of the Chapin diagram: when you have too many decisions operating at once, each one needs at least a little space on the line to show its individual flow path. The result is that you will have little room left on the line to state a particular step or action. This kind of problem is normally resolved by having the primary Chapin diagram point to one or more secondary diagrams where the individual choices and the appropriate actions are specified.

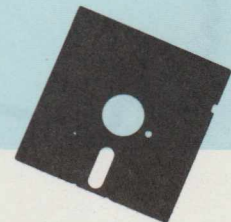
Sector 4: Choosing a Mapping Method

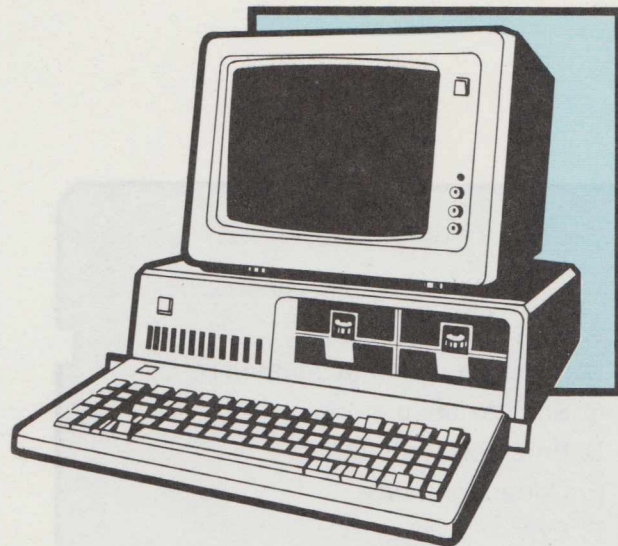
Each method of defining and charting the program flow has its own advantages and disadvantages. There is no single “ideal” mapping method, suitable for use with all kinds of programs. Normally, each programmer will find the method that is best suited to the kinds of programs he or she most commonly writes. Also, different programmers find different mapping methods more helpful than others in visualizing and developing the desired program behavior.

As we progress through this and future modules, we will use different techniques at different times. Later, as you design and develop programs of your own, you can use whichever method is best suited to you and to the kinds of programs you are working with.



NOTES:





TRACK 2



DETAILING THE SOLUTION PROCEDURE

If your computer is available at this time, turn it on as usual, and LOAD and RUN the program "TRACK2" from your Program Disk in Drive #8. Return to this point when the program instructs you to do so.

The primary requirement in designing the computer solution to any problem is to specify in detail all of the steps to be performed by the computer in the correct order. At the end of this procedure, the final flow chart, Chapin diagram, or other program flow map clearly defines the program structure and sequence.

However we don't want to start by drawing a flow chart or any other kind of diagram. The program outline we initially developed does not have enough detail in it yet to justify drawing any kind of program map. If we did draw a flow chart at this time, we would have to redraw it several times in the process of detailing the solution.

Accordingly, we will start with the outline itself, and add as much detail as we can to the list. Then we can use one or more mapping procedures to organize the solution procedure and to clarify choices and alternative program sequences.

Sector 1: Separating the Outline into Functional Units

Let's take another look at the basic outline for the checkbook register program. Figure 2-1 shows this outline. This outline is fine for the initial description of the program sequence, and it does show the functions we selected to put into the program. However it says nothing about how any of these tasks will be done. Also it does not yet contain all of the intermediate tasks that must be performed by the program in normal operation. The first step in accomplishing all of this is to rewrite the outline as a list of individual tasks for the program to perform.

1. Get today's date.
2. List available program options.
 - A. Enter new checks.
 - B. Reconcile monthly statement.
 - C. Scan existing entries.
 - D. Print selected entries.
 - E. Correct an entry.
 - F. Exit.
3. Request user input.
4. Execute user-specified routine.
5. Return to Step 2.

Figure 2-1. The basic sequence of events for the checkbook register program.

We will consider each task individually and define it in detail separately from all other tasks. Then the outline will help us to place the individual tasks in the correct order later.

Each individual task or program module goes on its own sheet of paper, as shown in [Figure 2-2](#). This allows us to easily compare modules with each other, and to change the order in which we want to place the modules. Also handling each module individually allows us to work on any module at any time, according to the requirements of the moment. Don't try to save paper at this point; you will only cause unnecessary confusion a little later on. Of course, this does not mean that you should waste paper. Neatness counts here, since a neat module description and detail expansion not only requires less paper, but is easier to read and understand as well.

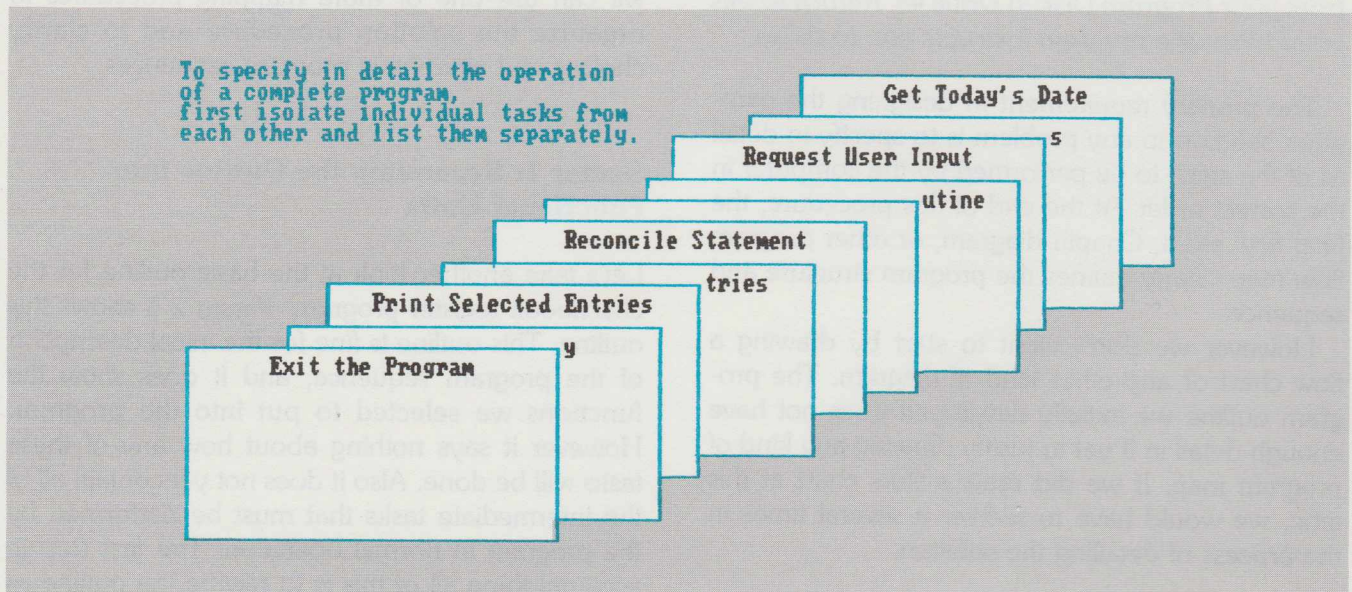


Figure 2-2. Separating individual tasks.

Sector 2: Detailing Individual Tasks

Once you have listed the various tasks individually, add detail to each task or program module by itself, as shown in [Figure 2-3](#). Each task or module will be a single logical unit within the entire program, and can be defined as such. Later, you will put the individual units together again in the proper sequence.

Of course, [Figure 2-3](#) doesn't really show all of the detail that will be required at this stage; we just used this much in our visual display on the computer because of space limitations on the screen. Since the figure is incomplete, let's look at the entire detailed procedure for exiting the program, and why the steps are necessary.

1. Optionally, place a message on the screen to indicate that the program is ending. This is more for the sake of appearance than anything else, and to provide a visual indication that the program has accepted your command and is doing something about it.
2. Get Record #1 from the file. This is necessary because we don't know ahead of time just where we will be in the file. We don't want to use just any record, because the specific fields we want may not be filled in, while other fields that we don't want may be filled in. We have already specified that Record #1 will be used to maintain information that must carry forward from one session to another. Also by reading

this record from the file, we make sure that all of the fields we don't want will be filled with blanks.

3. Put the current values for check number, per-check fee, and running balance into their appropriate fields in Record #1. These are the values that will be used to start out the next working session with this program.
4. Write Record #1 back to the disk. This action makes sure that the record will actually be updated on the disk itself, and not just in your computer's memory. Remember that anything in memory will be lost if the computer is turned off or another program is run, but data on the disk will remain there.
5. Close the file. This is very important. When you write to a disk file, the computer's operating system may or may not actually access the disk itself. Instead, it keeps small blocks of data in its own buffer space until it has enough data to fill a complete sector on the disk. This is necessary because the physical organization of data on the

Next, define each task separately, in detail.

Exit the Program
Update Record #1
 Last check number
 Current check fee
 Current balance
Close the file

Figure 2-3. Adding detail to a task.

disk is such that data *must* be read or written in sectors, rather than individual characters or records. Therefore, the system accumulates data until it can write an entire sector, or reads an entire sector and sends data to your program as required. When you close the file, the operating system writes to the disk any data that it still has in its buffer.

Of course, as you list the detailed operations to be performed within a given task, you will not normally include the reasons for performing each action. Just the statement of the step will be sufficient. If you are not sure whether or not a step will be required in a particular situation, include it in the list with a question mark. Later, when you are coding and testing the program, you can try it both ways and see what difference it makes. This procedure will also give you experience for your next program.

Sector 3: Adding Secondary Tasks to the List

As you work on any one program module or segment, you may think of another module that will have to go into the program, even if it did not appear in the primary outline. Whenever this happens, simply write the module requirements or function at the top of another piece of paper, and continue with the module currently in progress.

For example, the arrangement of characters on the screen often requires special consideration, which develops into some specialized tasks. One of these is the generation of the display outline itself.

This was your practice problem from Module 2, and we will look at it in detail later.

Another common problem is how to handle keyboard inputs within the framework of the existing video display. When we are filling in blanks on a display outline, we don't want to have a display line erased by BASIC when we press the <return> key. We must handle the keyboard separately from the display, and generate the display under program control, rather than just echoing characters from the keyboard.

Fortunately, almost all versions of BASIC permit this sort of isolation between the keyboard and the display, although the programming is slightly more complex. We will see how it can be done when we reach our "Practice Problems" section. For now, let's detail the requirements of such isolated keyboard entry, so that we will be able to develop the actual BASIC instructions later on.

Figure 2-4 shows the basic requirements. First, before any actual data is entered, we want to be

1. Use cursor keys (arrows) to select desired field. ONLY IF NOT NOW ENTERING DATA.
2. Left arrow or backspace should delete one data character and back up cursor within the field. ONLY IF ENTRY IS IN PROGRESS.
3. Any data key (letter or digit) begins or continues data entry in the present field.
4. Characters in excess of field length will not be accepted.
5. The <return> key terminates entry into the selected field. New data then replaces any previous data.
6. After <return>, routine ends and control returns to routine that called for keyboard entry.

Figure 2-4. Requirements for keyboard entry routine.

able to use the cursor keys (the arrows) to select the desired field on the display. However, once we have started entering data, we don't want to change fields, so the cursor keys should then be ignored.

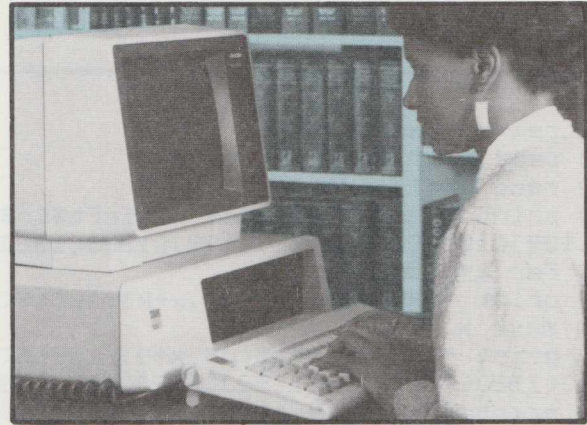
Once we have started to enter data, we want to be able to correct any errors, so we must be able to use the backspace or left arrow key to back up and delete a character so we can change it. This will be true as long as any data still exists to change; we don't want to find ourselves backspacing past the start of the field. If we try to backspace too far, it would be better to return to the field selection process. This will allow us to change our minds as to what field we want to use, as well as what we want to put into the field.

Any letter or digit key, punctuation, or space will either begin or continue data entry into the presently selected field. However characters will only be accepted until the selected field is full. Characters beyond the field length will be ignored.

The <return> key will be used to terminate entry into the current field. The data just entered will then replace any data previously in that field. This will allow a field to be changed when desired.

After the <return> key has been recognized and the field has been updated, keyboard entry is complete and should return to whichever routine called for keyboard entry.

Of course, we may wish to modify the keyboard routine later. For example, we might want to just have the keyboard routine return a character string, which the rest of the program can use for several purposes. This would mean that we would not want the keyboard routine to update the field; that



would be done by another part of the program. In any case, we aren't sure yet just how this will be handled, so we will leave the keyboard input routine alone for the present.

Sector 4: The Concept of Subroutines

There are many occasions in programming where the same task must be performed two or more times, often at widely separated points in the program sequence. If the task can be stated as a single instruction or as a very short sequence of instructions, then it can be placed in line with the program sequence at each point where it is needed.

Very often a more complex task, requiring a longer sequence of instructions, is required at more than one point in a program. In such a case, it would be nice if we could use the same instructions at all such points, rather than having to copy them at each point. Such a routine could then be called by other routines, and would return upon completion to the point from which it was called.

Such an arrangement does exist, and is supported by BASIC. Routines designed to run "under" other routines are known as *subroutines*. A subroutine can be called at any time by any routine that requires it. A subroutine can even be called by another subroutine.


```

800 rem box subroutine
810 read x:if x<0 then return
820 read y,c$
830 print "[HC]";left$(d$,x);left$(r$,y);"[176][192][192][192][192][192][192][19
2][192][192][192][192][192][192][192][192][192][192][192][174]";
840 for k=1 to 7
850 print "[HC]";left$(d$,x+k);left$(r$,y);"[221]";
860 next k
870 print "[HC]";left$(d$,x+8);left$(r$,y);"[173][192][192][192][192][192][192][1
92][192][192][192][192][192][192][192][192][192][192][192][189]";
880 print "[HC]";left$(d$,x+1);left$(r$,y+int((21-len(c$))/2));c$;
890 for k1=1 to 1500:next k1:goto 810

```

Figure 2-5. The subroutine to draw a box for the TRACK2 program.

In BASIC, the instruction to call a subroutine is GOSUB. It works just like GOTO, in that control is transferred to the line number following the keyword GOSUB. The difference is that with GOSUB, BASIC first remembers exactly where in the program the GOSUB statement appeared. Then the last statement in the subroutine is a RETURN statement. This tells BASIC to recall where it was when it encountered the GOSUB keyword, and to execute the next statement following the GOSUB command.

The program TRACK2, on the disk enclosed with this Learning Guide, contains an example of a subroutine. *If your computer is available, you should still have this program in memory. In this case, LIST the appropriate program lines to the screen and compare your program lines with the Commodore 64 and C-128 version that we have placed in the figures that accompany this discussion.*

Figure 2-2 showed all of the boxes drawn by the TRACK2 program for the IBM PC and compatible computers. Your display was similar, allowing for the number of rows and columns available on your display. *To review the manner in which these boxes were drawn, RUN the TRACK2 program again, just as you did at the start of this track.*

As the program draws the boxes on the screen, compare the boxes with one another. In what ways are they the same? How are they different from each other?

The boxes are the same in that each box is exactly the same size and shape as all the other boxes. They are different in that each box is placed at a different location on the screen, and the task name listed within each box is different from the others.

Figure 2-5 shows the routine that draws the boxes for the TRACK2 program. *Compare this program listing with yours by typing in the command:*

LIST 800-890 <return>

Note the RETURN command in line 810. This command indicates that we are indeed dealing with a subroutine, which must be called with a GOSUB statement.

Now type in the command:

LIST 1000- <return>

and observe the resulting display. If all of the lines don't remain on your display, don't worry. They all


```

1000 rem data for step 1
1010 data 1,18,"Get Today's Date"
1020 data 3,17,"List Options"
1030 data 4,15,"Request User Input"
1040 data 6,13,"Run Routine"
1050 data 7,11,"Enter New Checks"
1060 data 8,10,"Reconcile Statement"
1070 data 10,9,"Scan Entries"
1080 data 11,7,"Print Entries"
1090 data 12,4,"Correct an Entry"
1100 data 13,3,"Exit the Program"
1110 data -1
1200 rem data for step 2
1210 data 8,0,"Then",-1
1220 data 8,6,"add"
1230 data 9,0,"detail",-1
1240 data 16,7,"Update Record #1",-1
1250 data 17,10,"Last check",-1
1260 data 18,10,"Check fee",-1
1270 data 19,10,"Balance",-1
1280 data 20,7,"Close the File",-1
1300 rem data for exit
1310 data 21,25,"Now, return to"
1320 data 22,24,"Track 2 in your"
1330 data 23,24,"Learning Guide."
1340 data 22,0,"",-1

```

Figure 2-6. Calling the subroutine.

have exactly the same format, and you can see the pattern in the lines that remain on your display. Also, Figure 2-6 shows these lines from the Commodore version of this program.

To see how it works, first look at line 290. (Type `LIST 290<return>` to show this line on the screen.) This line simply calls the subroutine with a `GOSUB 800` command, and then delays for awhile with a `FOR...NEXT` loop.

When BASIC sees the `GOSUB` command, it first reads the line number to which it must go. Then it looks for either a colon or end-of-line marker to find the beginning of the next instruction. In this case, the next instruction is the `FOR` statement following the colon. BASIC saves the location of this instruction in a special area of your computer's memory, and proceeds to line 800, which is the destination line number of the `GOSUB` command.

Now looking back at the subroutine itself (Figure 2-5), we see that it starts by reading the row number at which the box is to start. If this

number is negative, the `RETURN` command will be executed. If not, the program continues to read the column (Y) where the box will start and the title (C\$) to be placed in the box. Line 830 draws the top line of a box 21 characters wide (including the corners) starting at that point.

The `FOR...NEXT` loop in lines 840 through 860 draws the sides of the box, seven rows high. Again, the values of X and Y are used here to put the sides in the right places. Then line 870 positions the cursor and draws the bottom of the box.

Line 880 places the title (C\$) at the top of the now-completed box. Since the length of the title will change from box to box, we use the `LEN` function to determine just how long the title is. This allows the program to center the title on the top line of the box. Then, line 890 delays briefly, after which it uses a `GOTO` command to start the subroutine over at line 810.

When the `READ X` command finds a -1 value, it is the signal to execute the `RETURN` command.

When BASIC sees this instruction, it goes back to its special area of memory to find out where its last GOSUB came from. It then resumes execution at the point where it left off (in this case, the FOR statement in line 290).

When the next GOSUB is executed in line 310, it will call another subroutine at line 900. This subroutine will read row and column values as before, but will not draw any boxes. Rather, it will place a character string at any desired location on the screen. It will RETURN when it reads a negative X value.

The subroutine at line 900 will place additional character strings on the screen each time it is called. Any characters in the specified locations will be overwritten, but the rest of the screen will not be disturbed.

The advantage of using a subroutine is that the same set of instructions can be used any desired number of times to do a given task. The same subroutine can be called from anywhere within the program, and it will do the same thing each time it is called. The only requirement in using the subroutine is that any changing data must be passed to the subroutine. The subroutine will then act on this data in the same fashion each time.

When you detail the behavior of a subroutine, treat it as any other routine in the overall program. Simply note "Return to calling routine" or a similar action as the last action to be performed by the subroutine. In the main routines, note any point where the subroutine is needed as "Call subroutine xxxxx", where "xxxxx" identifies the particular subroutine.

A program may contain any desired number of subroutines, and one subroutine can call another one if needed. This last process is known as "nesting subroutines" one inside the other. The number of subroutines that may be nested at any one time (the number of GOSUBs before BASIC encounters a RETURN) is determined by the amount of memory available to BASIC to remember the points to which it must return. Subroutines always operate on a last-in, first-out basis. That is, the first RETURN statement encountered will return control to just after the *last* GOSUB statement executed. Figure 2-7 illustrates this process.

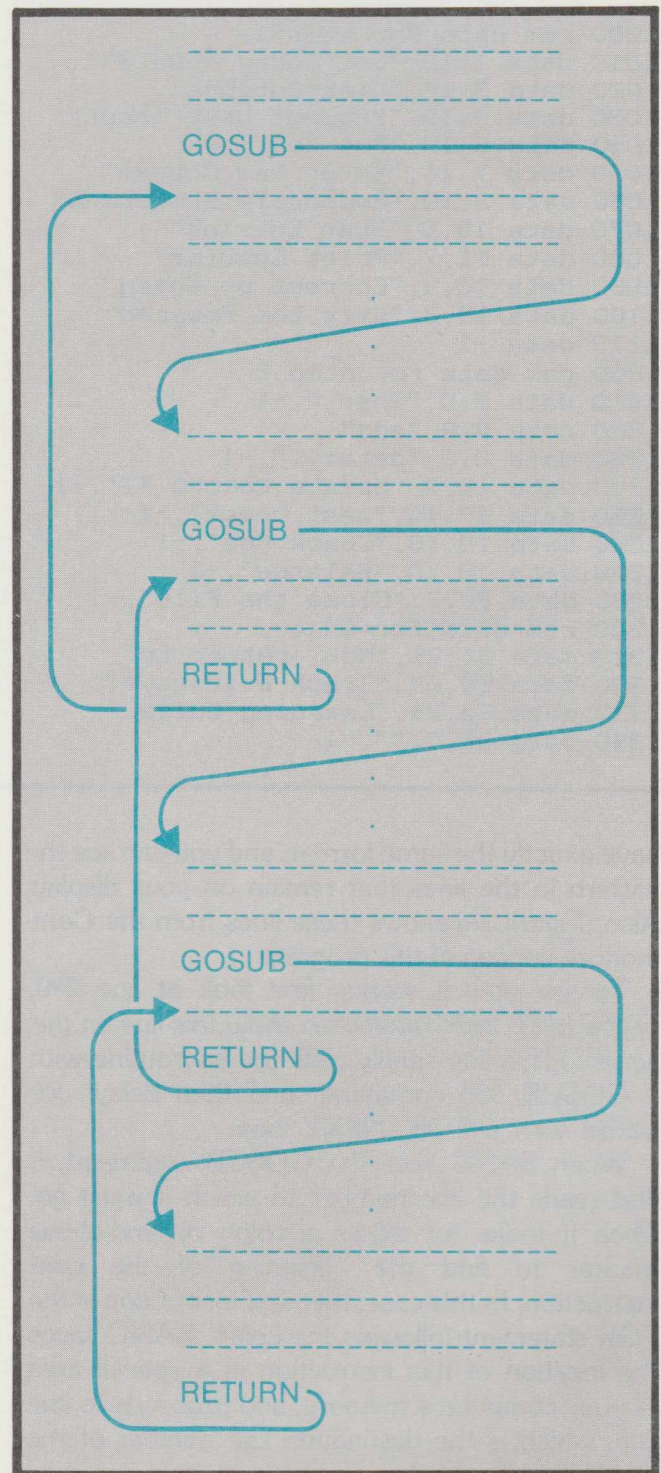


Figure 2-7. The behavior of nested GOSUB and RETURN commands.

TRACK 3



DRAWING FLOW CHARTS

Once you have detailed each module or working section of the program you are designing, it is time to rewrite the solution procedure in a pictorial form. In this way, you can present the solution procedure in a format that clearly shows all possible choices and decisions, and program flow paths that can result from each decision or choice.

One of the most common types of pictorial display is the flow chart. The flow chart is so named because it charts the program flow from start to finish in a pictorial manner that is explicit, clear, and easy for others to follow. We took a brief look at flow charts in Track 1 in this Learning Guide. Now we will look more closely at flow charts, and see some of the many ways in which they can be used.

If your computer is available at this time, turn it on and insert your Program Disk. Then, type in the commands:

```
LOAD "TRACK3",8<return>
RUN<return>
```

Return to this point when the program instructs you to do so.

Sector 1: Flow-Charting Symbols and Rules

In order to use flow charts effectively, you must draw them so that you will clearly understand them at some later time, and so that another programmer will be able to follow them easily. To help make this possible, a set of standard flow-charting symbols and a set of flow-charting rules have evolved, so that anyone familiar with standard flow-charting procedures can read and use one drawn up by anyone else who used the standard procedures.



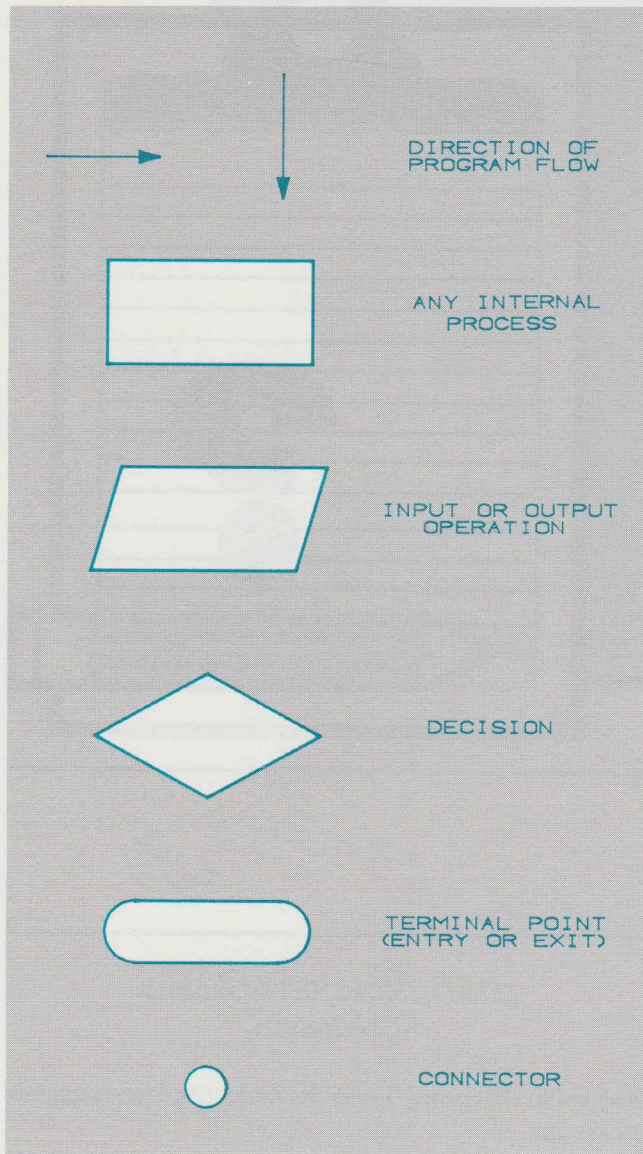


Figure 3-1. The primary flow-charting symbols.

Figure 3-1 shows the primary flow-charting symbols. These are the main symbols used in programming applications, and in fact you can diagram any program using just these symbols. However in many cases it is convenient to use some additional symbols to clarify the nature of certain kinds of operations. Thus, you may find it useful to use the secondary symbols shown in Figure 3-2, when they are called for.

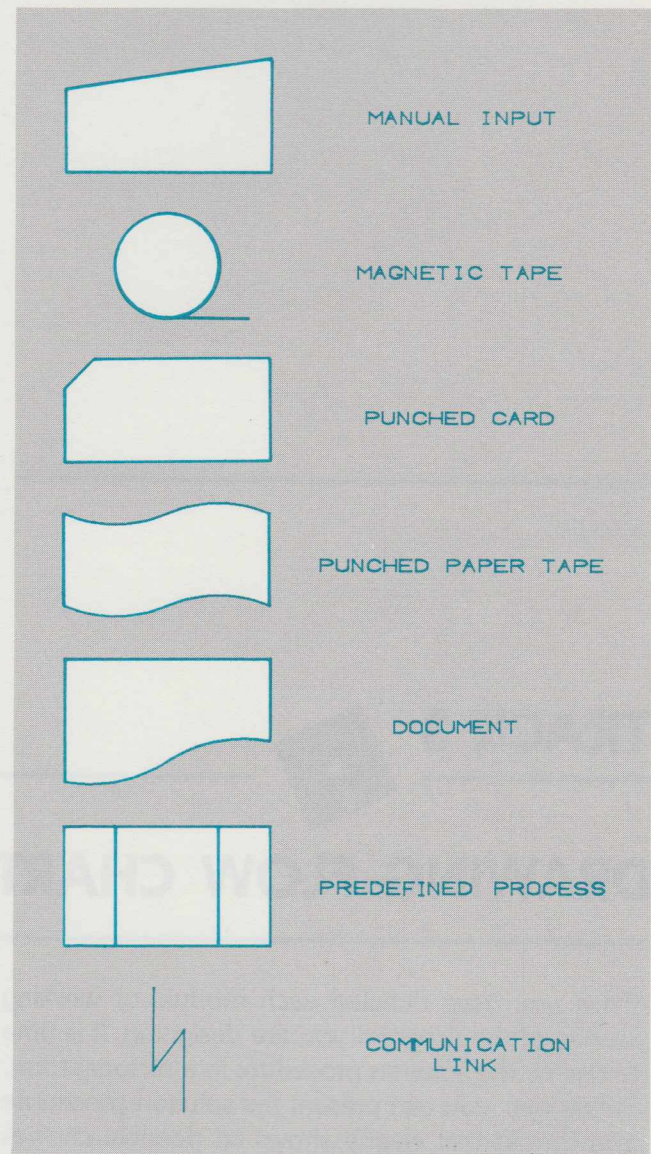


Figure 3-2. Secondary flow-charting symbols.

The secondary symbols in Figure 3-2 are not commonly used in the actual programming process. However flow charts are commonly used to describe many different kinds of operations and activities; they are not restricted to just programming applications. You may see flow charts used to describe any kind of process, and you should recognize the more commonly used symbols in such applications.

Flow charts are always drawn in accordance with specific rules, which help to ensure that the chart is clear and easy to follow. We will adhere to these rules in this and all future modules, and you should adhere to them whenever you draw a flow chart of your own. These rules are logical and easy to follow, and help to avoid misunderstandings later on, when you or another programmer must follow the flow chart. Let's look at each of these rules:

1. Show program flow first from top to bottom, and then from left to right. Flow charts are basically organized vertically, starting at the left side of the page. Draw the entire first column, and then move to the right to draw a second column. Continue this pattern as necessary.
2. Label each box or symbol. Each symbol in the flow chart has a specific meaning that defines its purpose. However this meaning does not define the exact action or purpose of this symbol at that point in the program. Write the exact process, decision, etc. either inside the appropriate box or else beside it. For decisions, identify each possible decision outcome and appropriate outgoing flow line.
3. Use lines to connect the flow chart symbols, showing the exact path of program operation. Place arrowheads as needed to clarify the direction of program flow on each line. Where multiple lines come together to a common point in the program, place arrowheads on all lines approaching that point. Bring the lines together between symbols. Do not have multiple input lines to a flow chart symbol. (Multiple output lines from a decision are permitted.)

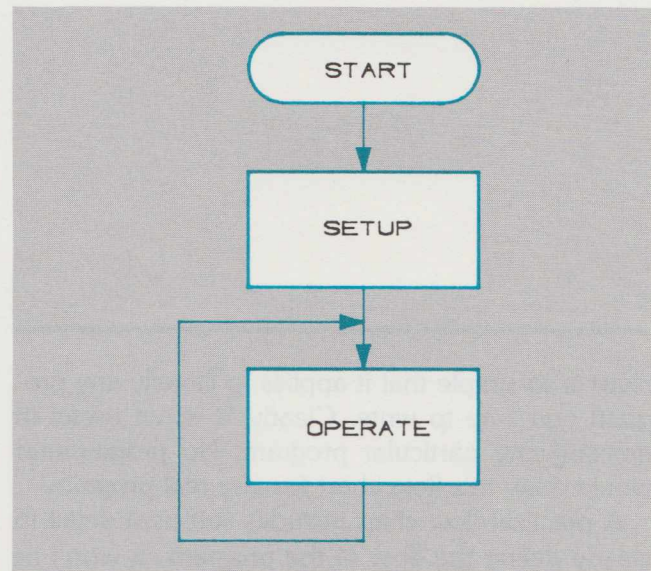


Figure 3-3. A very simple flow chart.

4. Do not allow lines to cross over each other. The chart should be kept uncluttered. If long lines or crossovers are required, use connector symbols instead. Place a number inside each connector symbol to indicate which connectors belong together.
5. Chart different routines on different pages. Each routine, subroutine, and procedure should have its own page. Then the main routine is placed on its own page on top. Each call to a subroutine or procedure is simply indicated as such on the main chart. Then the detail for the selected subroutine is provided on its own separate chart. In this way, each program module appears on its own page as a single logical sequence. This is much easier to follow than multiple procedures on one page.

By following these basic rules, you can readily create clear, concise flow charts that will be meaningful and useful to yourself and to other programmers.

Sector 2: Flow-Charting the CHECKBOOK Main Program Loop

As shown in Figure 3-3, the initial flow chart for any program can be quite simple. In fact, this flow

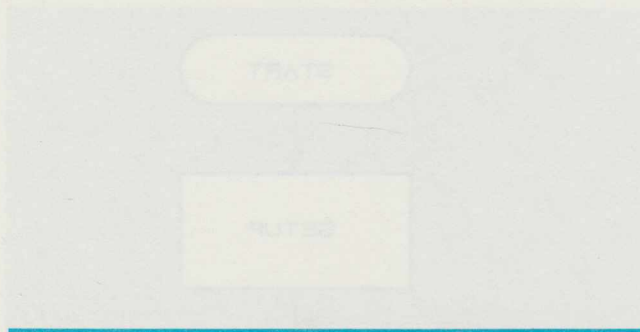


chart is so simple that it applies to literally any program you care to write. Clearly, it is not useful in defining any particular program. No programmer would draw this flow chart for any real program.

A practical flow chart includes sufficient detail to clearly define the flow of the program. It won't be all in one piece. As we mentioned before, each logical program module or block will have its own flow chart. Then there will be an overall flow chart that will simply identify each block and show its place in the program flow sequence.

To see how this works, let's draw a flow chart of the main body of the CHECKBOOK program. Figure 3-4 is a repeat of the basic program sequence outline. Remember, we will not chart any specific routine yet; we must start with the main body of the program. Later we will flow-chart the individual working routines.

1. Get today's date.
2. List available program options.
 - A. Enter new checks.
 - B. Reconcile monthly statement.
 - C. Scan existing entries.
 - D. Print selected entries.
 - E. Correct an entry.
 - F. Exit.
3. Request user input.
4. Execute user-specified routine.
5. Return to Step 2.

Figure 3-4. The basic sequence of events for the checkbook register program.

Figure 3-5 shows a more useful flow chart for the CHECKBOOK program. This is the flow chart that was drawn by the program TRACK3 on your program disk. Of course, the video display characters

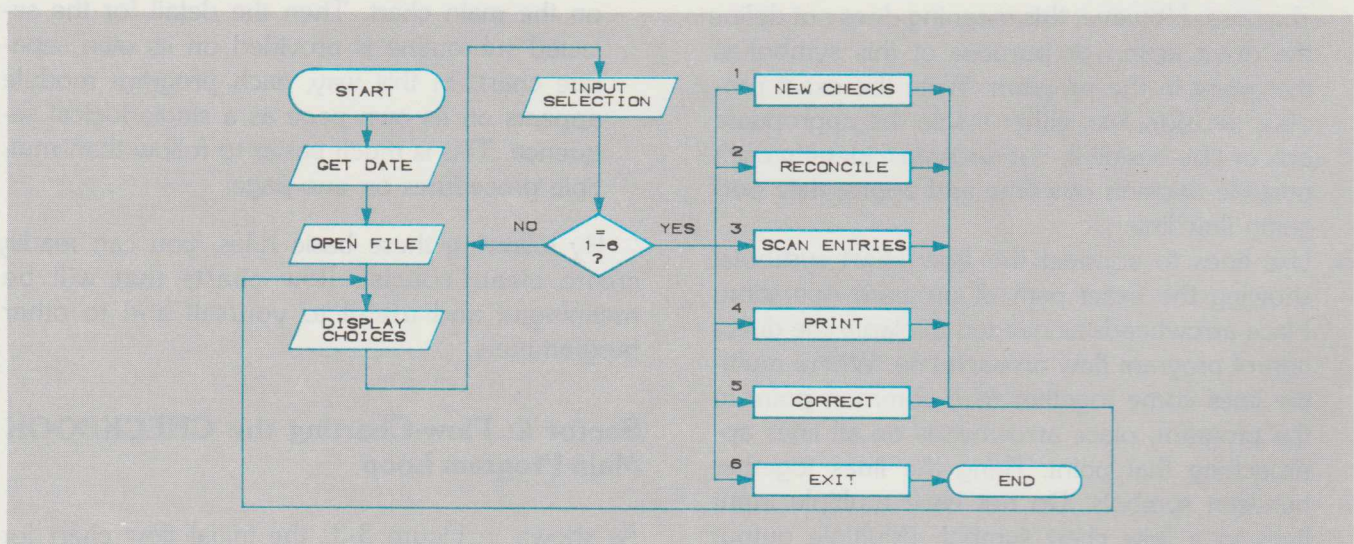


Figure 3-5. The overall chart for the CHECKBOOK program.

available on your computer do not include specialized flow-charting symbols, but the existing characters do permit the generation of a readable, though small, flow chart.

If your computer is available and the TRACK3 program is still in memory, type in the command:

RUN<return>

If you turned off your computer earlier, turn it on again and type in the command:

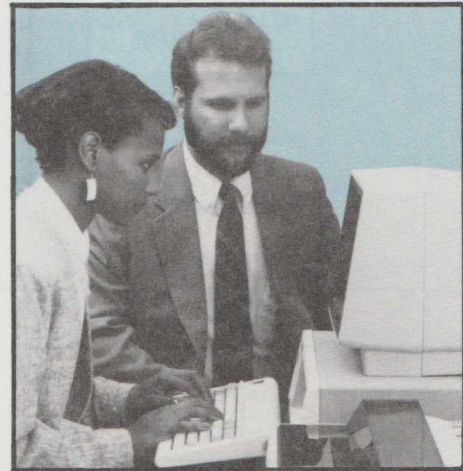
LOAD "TRACK3",8<return>

before typing in the RUN command.

Observe the generation of the main flow chart at the end of the program. Return to this point when the program directs you to do so.

In drawing this flow chart, the first column was direct and straightforward. We simply started at the top and drew in each general operation in turn, each in its own box. We used I/O (Input/Output) boxes for the first four actions, because they all involve some sort of interaction with the user. The detailed flow charts corresponding to each box will of course identify each operation and process specifically.

Following the display of the program choices, we show the first decision. This is essentially a test to make sure that the selected option is valid. If not, the selection is ignored and a new selection requested. If so, the program will advance to one of six possible routines, depending on the selected input. In this overall flow chart, only the name of each routine is given. As with the initialization routines, details of each option routine will be charted on a separate page.



Sector 3: Flow-Charting an Internal Routine

The flow chart for any inside routine within the program is drawn according to the same rules as the main program chart. The differences are first that we will deal only with part of the program, and second that we will incorporate more detail into the subchart than we did in the overall chart.

Figure 3-6 shows the first-round flow chart for entering new checks and other information into the CHECKBOOK register file. This flow chart still does not include all the detail that will be required to actually code the program, but it does clearly show what this routine is intended to accomplish. Before we are done, however, we will have to add routines to generate the desired display and to accept keyboard entries without disturbing the existing display, as well as to recognize special payee names as signals for functions other than checks (deposits, automatic deposits and withdrawals, electronic teller activities, etc). The first two of these will have to be general to all or most of the routines, while the third may be specific to this one routine. Either way, all three routines will have to be defined and charted in more detail eventually.

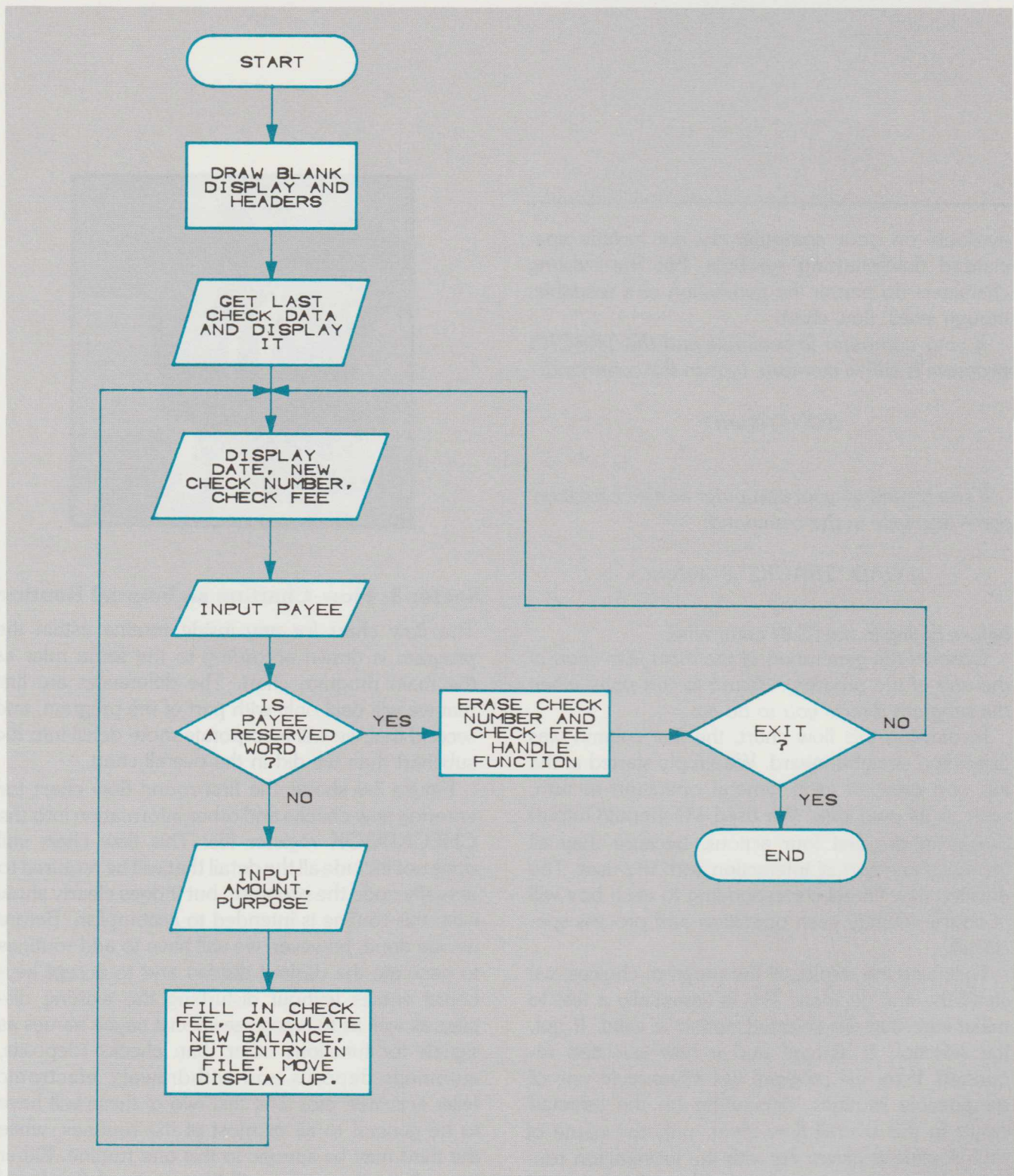


Figure 3-6. Flow chart for entering new checks into CHEKBOOK.

We don't want to define these three routines yet. First, we will define each of the main option routines. Then we will know which operations are common to two or more option routines and should be written as subroutines, and which ones are used only by a single option routine and therefore should be written as part of the option routine.

To determine how to write each internal operation, we first chart each option routine. For example, Figure 3-7 shows the option flow chart for the RECONCILE option routine. Similarly, Figures 3-8 through 3-11 (on the following pages) show the flow charts for the remaining option routines in the CHECKBOOK program.

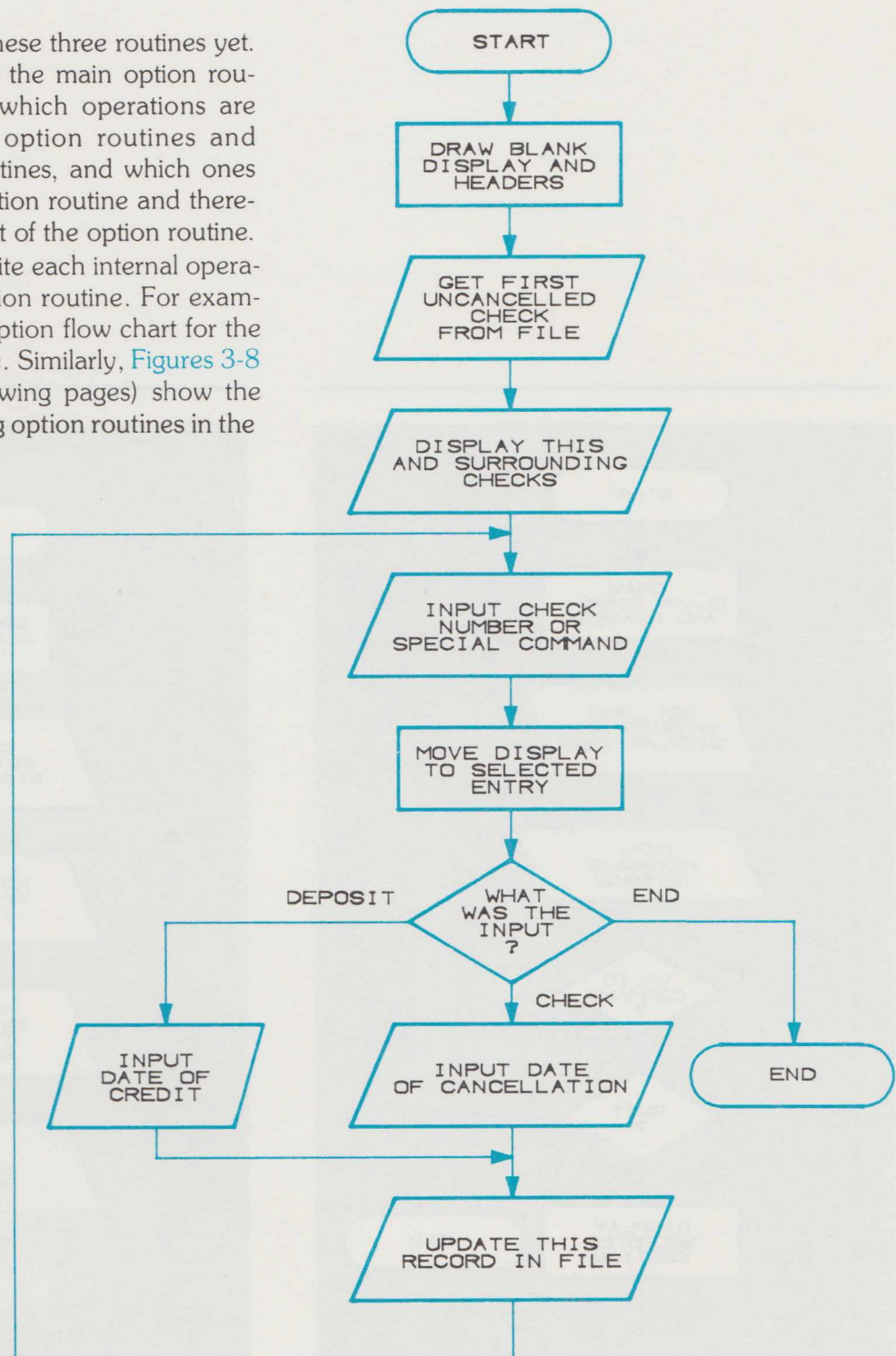


Figure 3-7. Flow chart for RECONCILE routine.

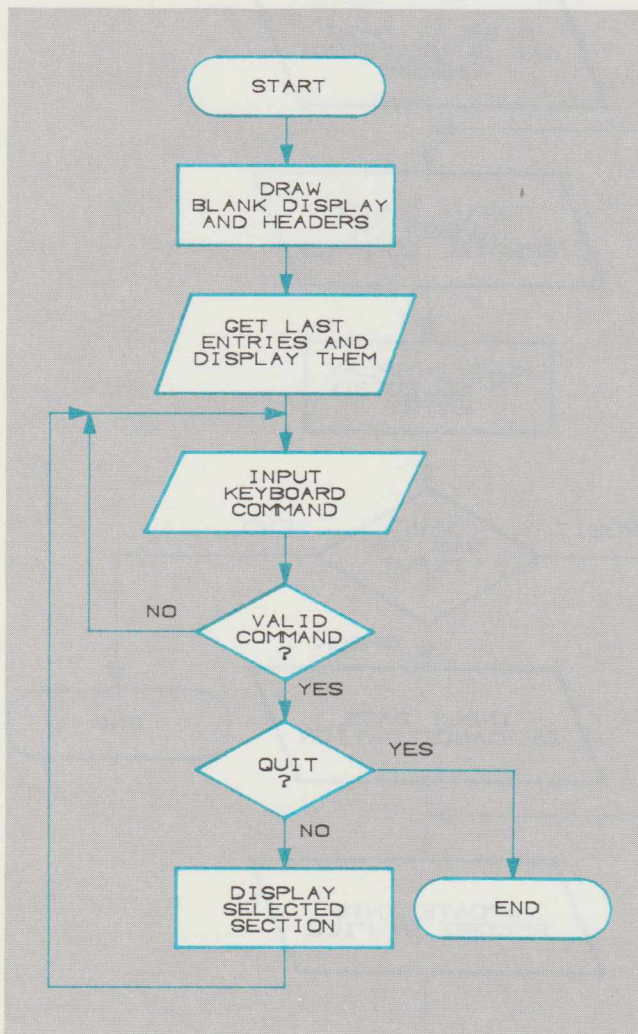


Figure 3-8. The flow chart for the SCAN option routine.

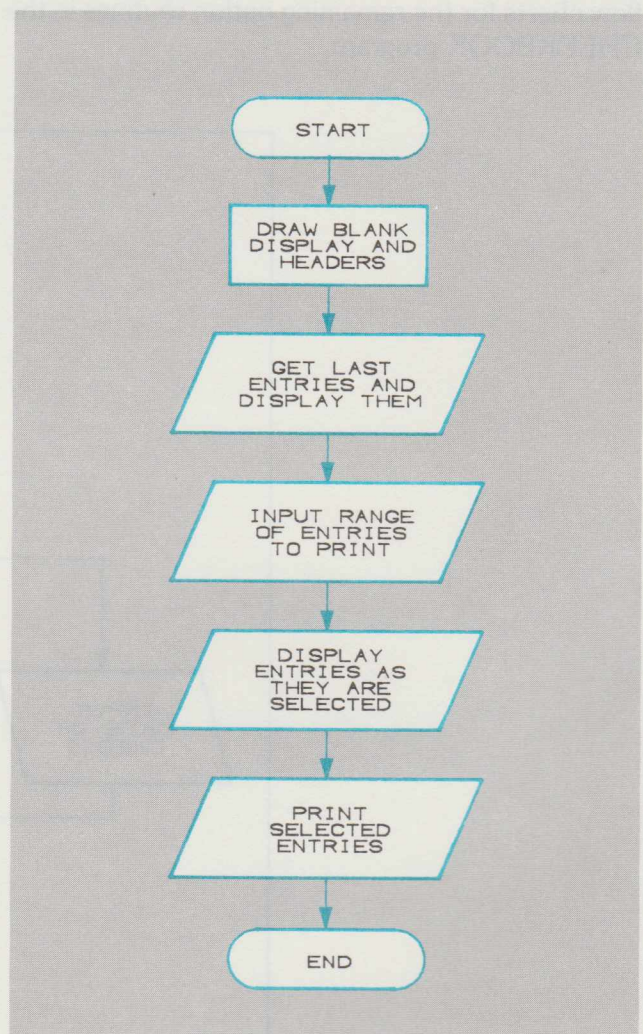


Figure 3-9. The PRINT routine flow chart.

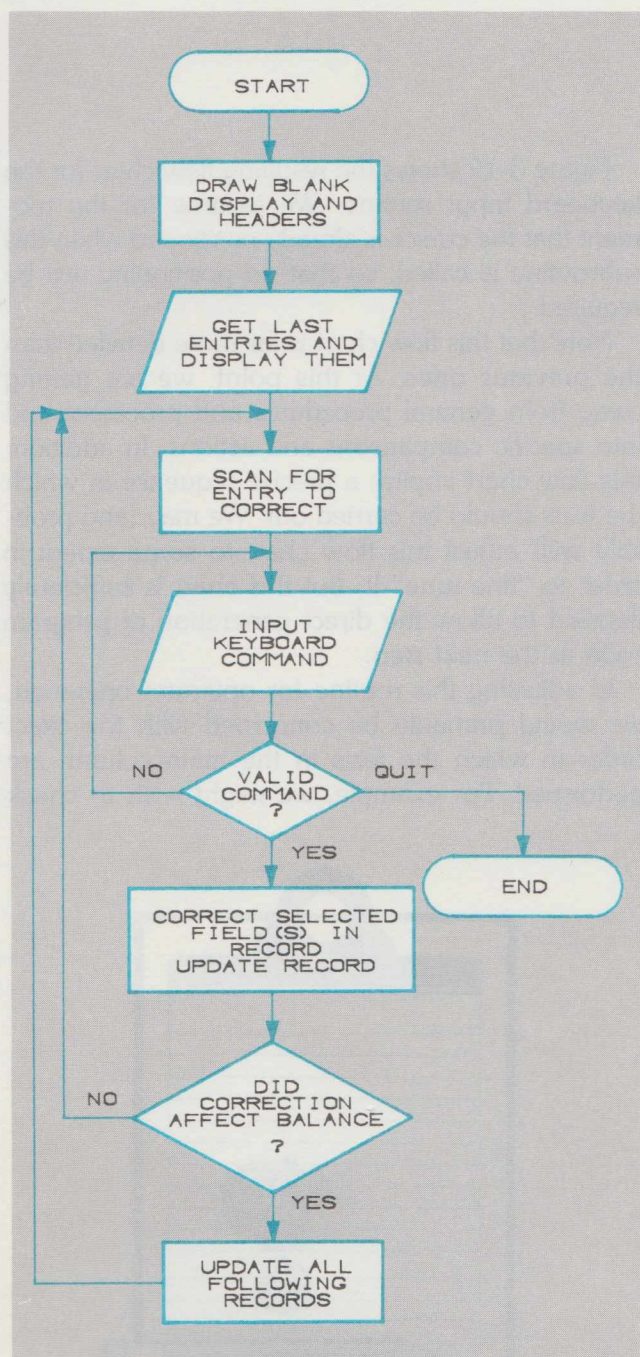


Figure 3-10. Flow chart of the entry correction routine.

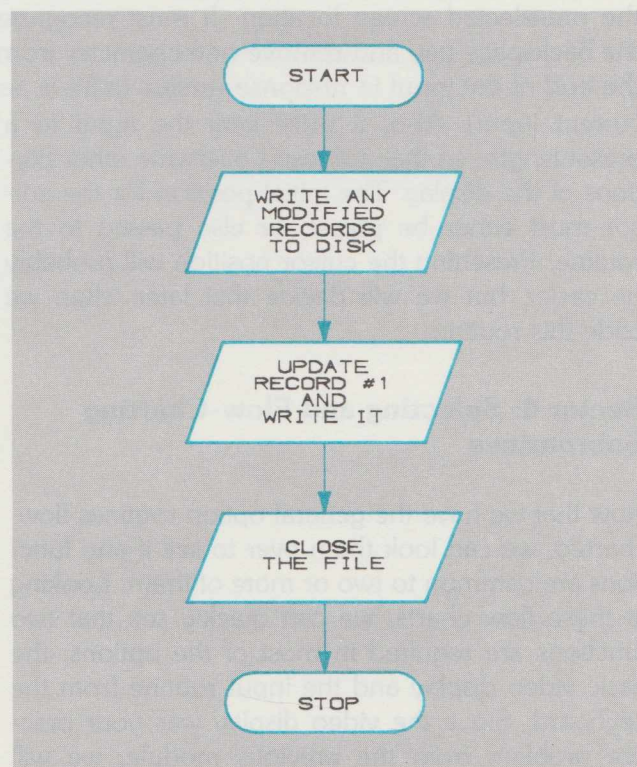


Figure 3-11. The EXIT routine flow chart.

The keyboard input routine will have to accept keyboard characters in either upper- or lower-case format and display these characters in order at the preselected screen location. It must recognize the backspace key and remove one character from the end of the input in response (unless there is no current input). Also, it must limit the input to a preset length, so that it doesn't overwrite other portions of the display. The initial position for the cursor must either be preset, or else passed to the routine. Presetting the cursor position will probably be easier, but we will decide that later when we code this routine.

Sector 4: Selecting and Flow-Charting Subroutines

Now that we have the general option routines flow-charted, we can look them over to see if any functions are common to two or more of them. Looking at these flow charts, we can quickly see that two functions are required in most of the options: the basic video display and the input routine from the keyboard. Since the video display was your practice problem from the previous module, we will save that until later. In this sector, we will chart the keyboard input routine.

Figure 3-12 shows the resulting flow chart for the keyboard input routine. We assume for the moment that the cursor is already positioned when this subroutine is called, so that no positioning will be required.

Note that this flow chart is far more detailed than the previous ones. At this point, we are getting away from general procedures and processes and into specific comparisons and actions. In addition, this flow chart implies a precise sequence in which the tests should be carried out. We may (and probably will) adjust this flow chart to some extent in order to "fine-tune" it, but this chart is sufficiently detailed to allow the direct generation of program code as the next step.

In adjusting this routine for optimum operation, we would primarily be concerned with the exact order in which the tests in the main column are performed. For example, we might wish to check



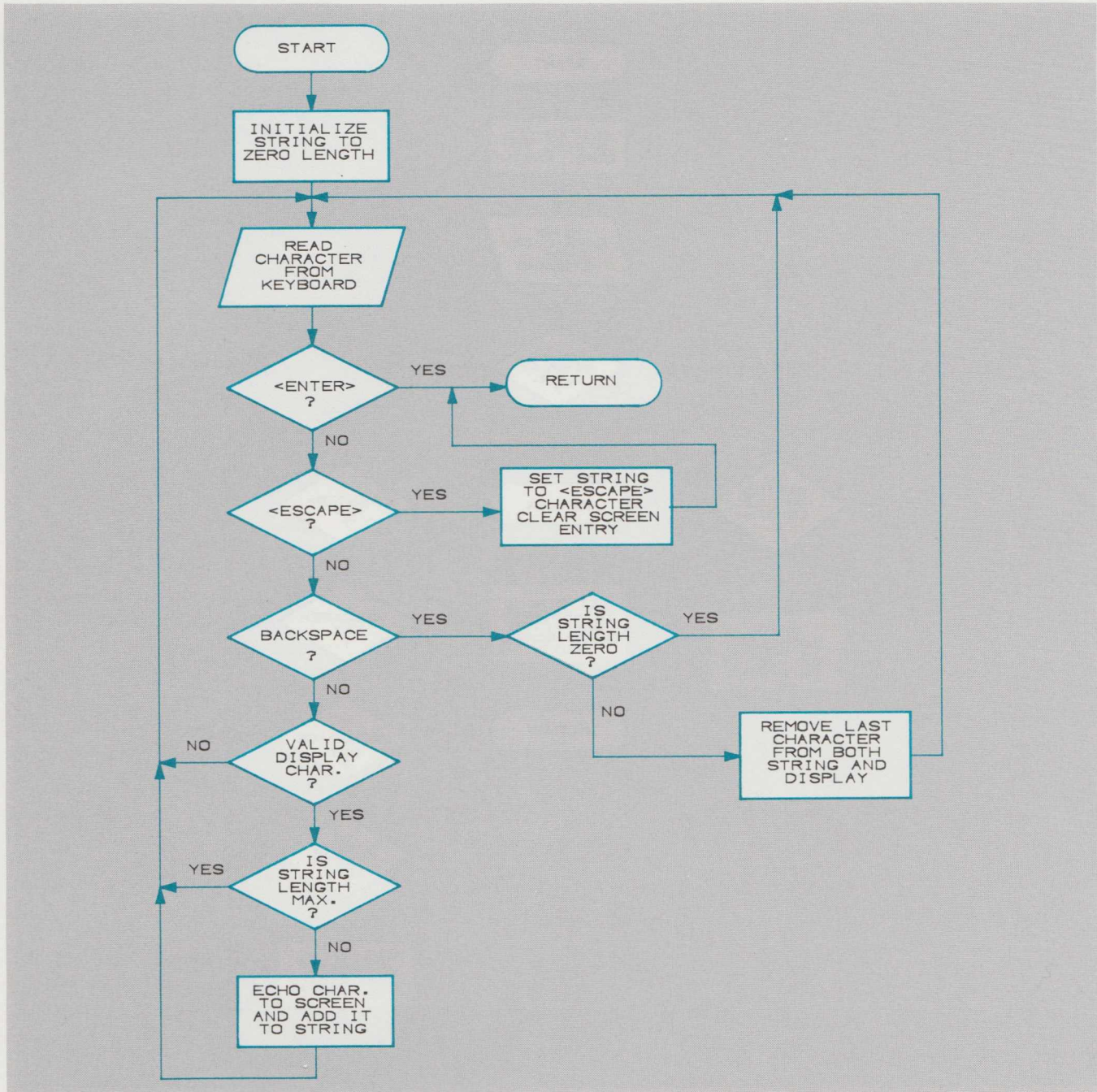


Figure 3-12. The flow chart for the keyboard input subroutine.

first for a valid display character, and then either check the present string length or else check for special control characters, as shown in Figure 3-13. The advantage of this is that the number of tests required for any single character entry will be reduced, so processing will be faster.

In setting up this flow chart, we have allowed for three special control characters. These are the <enter> key (to signify the end of a normal entry), the <backspace> key (to cancel the last character entered), and the <escape> key or an equivalent key, depending on the specific keyboard for a given

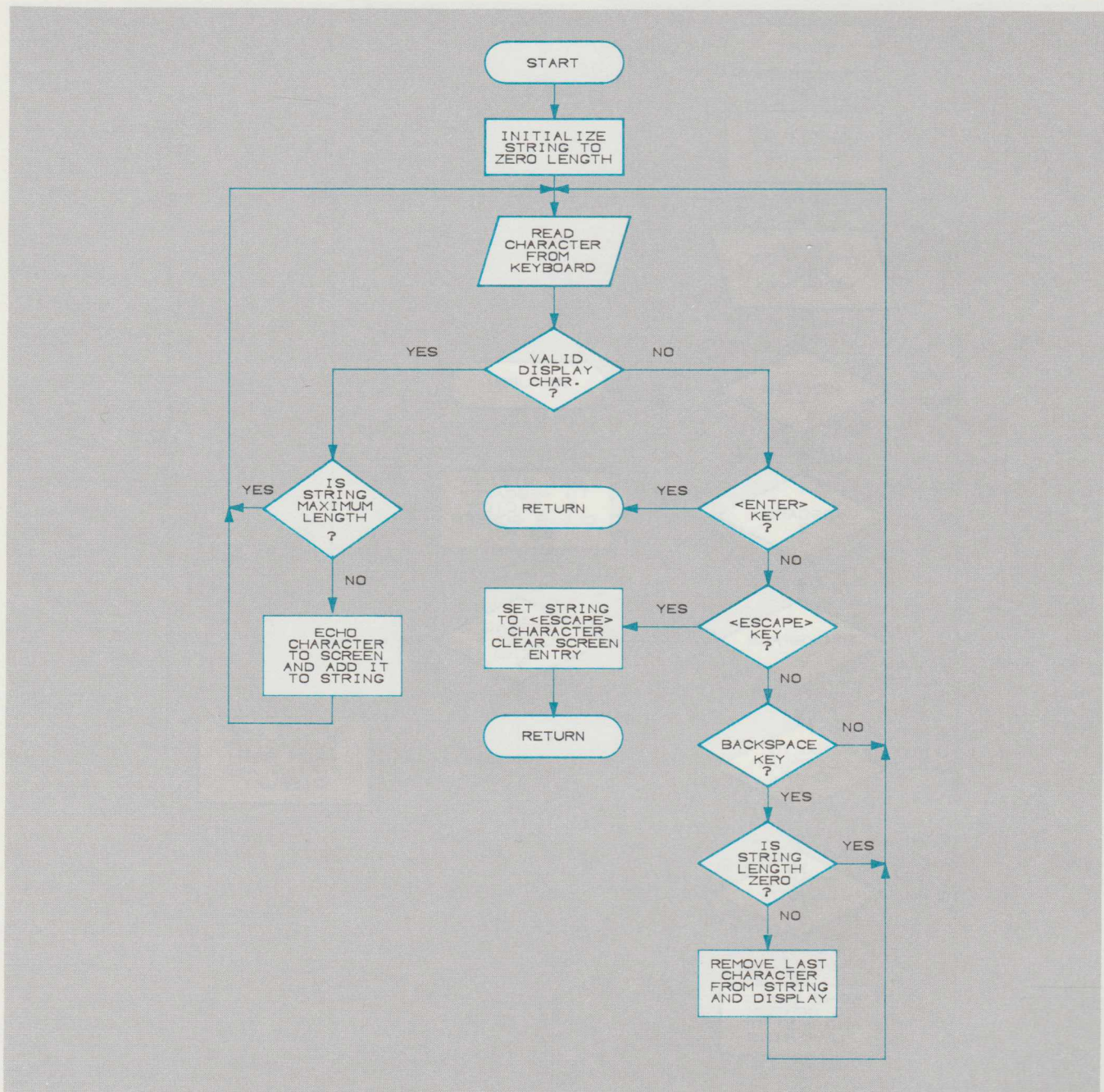


Figure 3-13. An alternate operating sequence for the keyboard input routine.

computer (to cancel the entire entry). If we think of any more requirements for this routine, we can use additional control characters to control the extra actions.

As we look over the general option routine flow charts for the CHECKBOOK program, we see that

one constant requirement will be to fill records into the already drawn video display. The blank display is your practice project, but the actual data will have to be filled in separately. The requirement for filling in the display appears in all except the EXIT routine, so it is a ready candidate for a subroutine.

Note that we do not want to make it just a part of the initial display subroutine, since the program will require that data be changed on the existing display, and not only be generated once. We don't want to have to redraw the entire display just to update it. Therefore, we will design the update routine as a separate subroutine.

For this subroutine, we will need to be able to relocate the cursor to place the various information on the screen. Therefore, we will need to supply the initial cursor position to the subroutine, which will then locate display characters relative to that position on the screen. In addition, we will have to already have the appropriate record from the disk file in the buffer.

Figure 3-14 shows the flow chart for this routine. As with the keyboard routine, this display routine is fully detailed at this level of flow-charting, and is ready for coding as the next step. The original screen position supplied to the subroutine is assumed to be the upper left-hand corner of the display section for the selected display record. Position is indicated as row (R) and column (C) for simplicity. Adjustments to this location are based on the display format we suggested in Module 2. If you wish to use a different format, you will have to adjust positions of individual entries accordingly. Also, we may later have to adjust the locations slightly to allow for errors in position assumptions. We will make any appropriate corrections when we test the program.

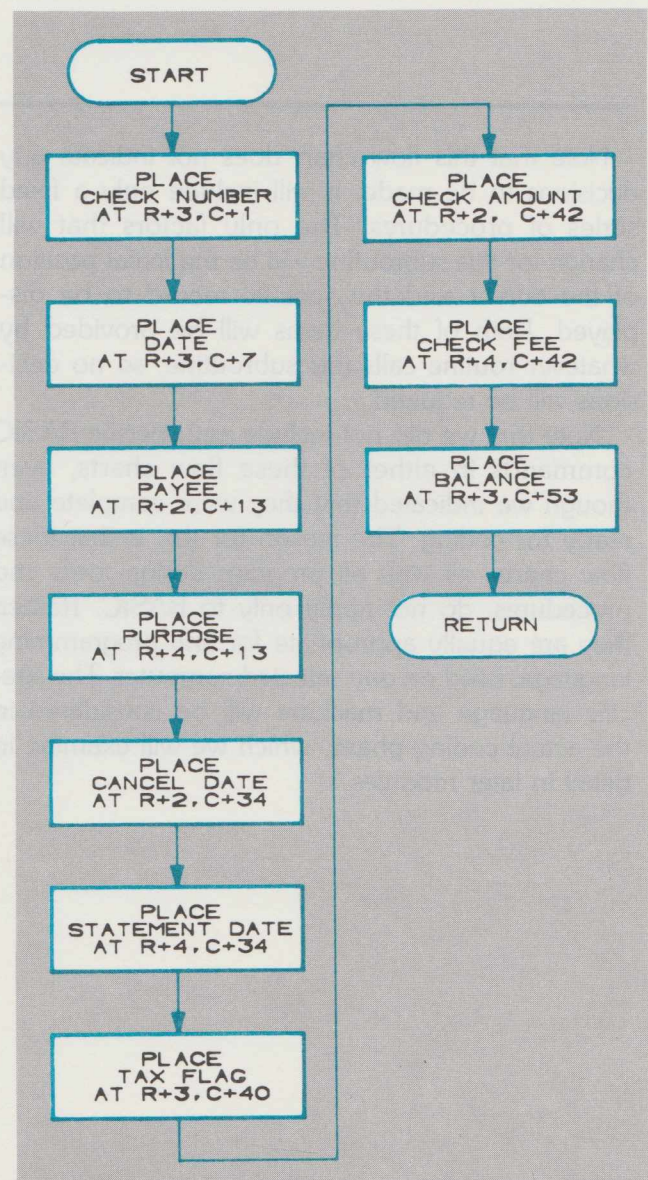


Figure 3-14. The flow chart for the single-entry display update routine.



Note that this flow chart does not indicate any decisions to be made; it will include only a fixed series of procedures. The only factors that will change for this subroutine will be the initial position of the cursor and the specific record to be displayed. Both of these items will be provided by whatever routine calls this subroutine, so no decisions will be required.

Note that we did not include any specific BASIC commands in either of these flow charts, even though we indicated that they were complete and ready for coding. The reason for this is that these flow charts, as with all program design tools and procedures, do not apply only to BASIC. Rather, they are equally appropriate for any programming language, used on any selected computer. The specific language and machine will be considered in the actual coding phase, which we will examine in detail in later modules.

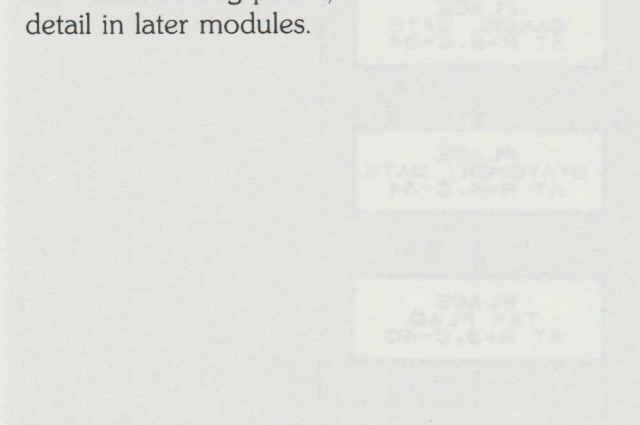
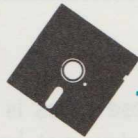


Figure 3-14 The flow chart for the subroutine display update routine

TRACK 4



THE CHAPIN DIAGRAM

For a long time the flow chart has been the “standard” method of depicting a task in graphic form. It clearly identifies each task by name and nature, as well as the sequence in which tasks are performed. Unfortunately, it also has a few drawbacks and limitations.

One of the main limitations of the flow chart is the amount of space it occupies. The spacing between individual boxes and symbols does make each box easy to pick out, but it uses up a large amount of space on the paper. In fact, in a clearly drawn flow chart, the amount of empty space, lines, and arrows between boxes will greatly exceed the amount of space actually used by the boxes themselves.

A second drawback is that the box limits the amount and format of the text that can fit into the box. Slanting lines especially limit the amount of text that will fit on a line, and such shapes as decision boxes have a hard time containing enough text to define the decision. This is sometimes overcome by placing the text next to the box rather

than inside the box. Unfortunately, this practice becomes even more wasteful of space on the page.

For some purposes, the flow chart is perfectly satisfactory or even ideal. Other applications, however, require a more compact format which also permits a more usable text format. One such format is the Chapin diagram.

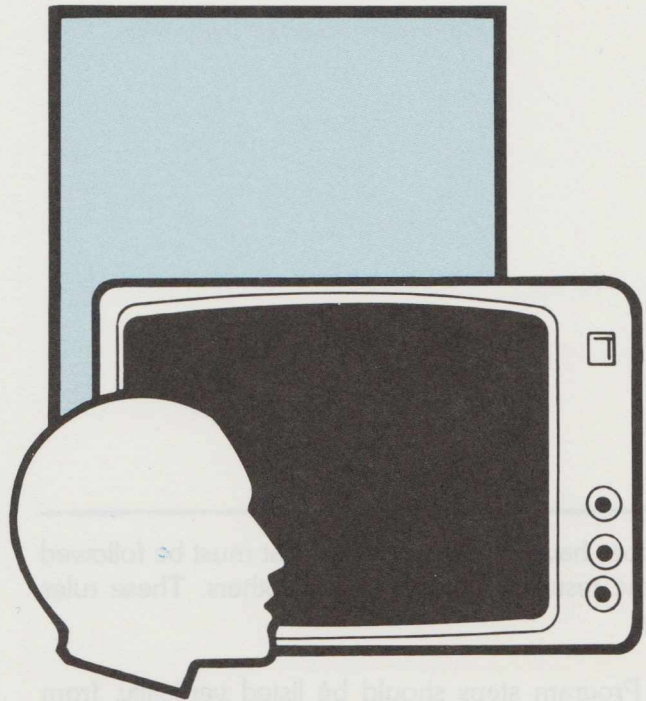
If your computer is available, turn it on if necessary and type in the commands:

```
LOAD "TRACK4",8<return>
RUN<return>
```

Return to this point when your computer instructs you to do so.

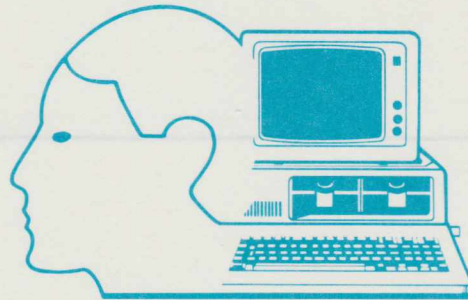
Sector 1: Rules for Using the Chapin Diagram

Although the Chapin diagram does not use special symbols and shapes like the flow-charting symbols,



it does have a few basic rules that must be followed if the result is to be legible to others. These rules are:

1. Program steps should be listed vertically, from top to bottom. This is similar to the primary sequence for flow charts, except that for Chapin diagrams there are no exceptions permitted to this rule. A return from the end of a repeating loop back to the beginning is the only bottom-to-top flow permitted.
2. No horizontal motion is permitted on the diagram. All program flow is from top to bottom, and there is only one main column of steps on the page. Alternate actions stemming from a decision may be listed side by side, but they are separate and independent actions; one column cannot lead to another. If an action is common to both alternatives, it must appear vertically under both alternatives.
3. Flow paths for program loops are commonly indicated on the left side of the diagram. Such loops are the only kind of flow path that may go from bottom to top. Normally, vertical paths indicating program loops should be marked with slanting lines or crosshatches to denote the reverse direction of logical program flow.
4. A single program module should fit on one sheet of paper. This does not mean that the entire program should be detailed on one page, but that each page should be a complete entity in itself. If a task spills over onto a second page, it is too complex and should be broken up.



The purpose of these rules is to make the programming process easier, not harder. The Chapin diagram lends itself to the design of structured, self-contained program modules which can be readily defined and corrected as necessary. This is not to say that larger modules cannot be written; they can be. In fact, the entire program is considered to be a module in that sense. However, in the top-level diagram, each function is identified only by a procedure name. The details of the procedure or module are listed on another page. Then that detailed procedure is placed on a single line of the main diagram.

These subprocedures may also contain working modules that require additional pages for detailed description, and so on. In any given diagram, however, each one-line entry will identify either a single program step or a complete submodule. No partial modules should appear on any Chapin diagram.

Sector 2: Diagramming the PHONEBOOK Program

Drawing a Chapin diagram is similar in many ways to drawing a list of program actions or objectives, such as we did in earlier modules. The primary

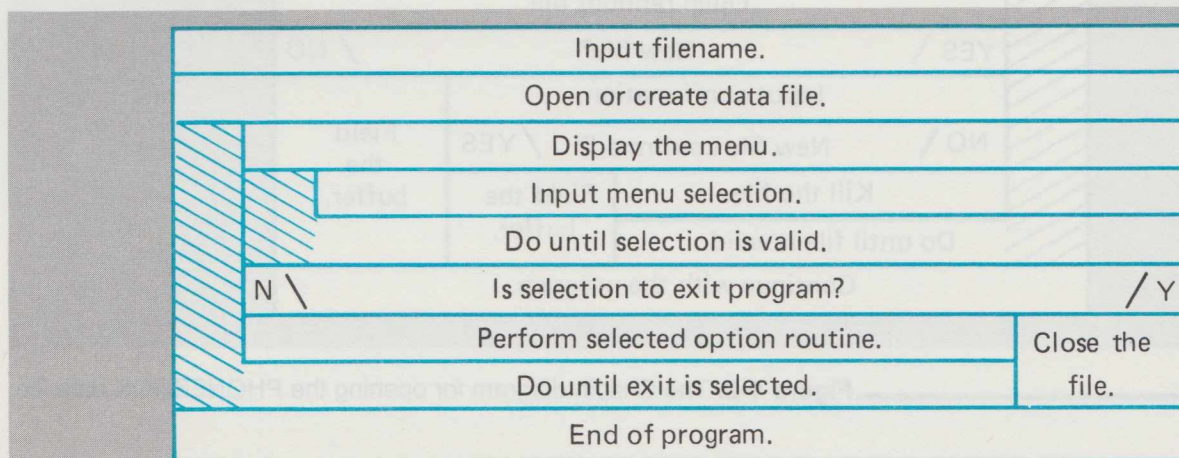


Figure 4-1. The first-level Chapin diagram for the PHONEBOOK program.

difference lies in the handling of decisions. [Figure 4-1](#) shows the general program diagram for the PHONEBOOK program we first examined in Module 2. Due to the limitations of some screen displays, and to help clarify the logic of the diagram, the computer-generated diagram was a simplified version of [Figure 4-1](#). This is also true of other Chapin diagrams in this track.

Although the computer program, TRACK4, on the enclosed program disk draws the general framework for the Chapin diagram as a first step, this is not generally the way it is done. The diagram is commonly drawn on standard lined binder paper on a step-by-step basis. The lines separating individual steps are drawn to fit, following each individual step. In this way, the width and height of the diagram are "customized" for this particular task, and are not predetermined.

As you can see from [Figure 4-1](#), each nested decision and program loop reduces the working width of the diagram, because of the need to put two items side by side on a single line of paper. Decisions are generally more space-consuming than program loops, especially if there are two independent activities of which one must be carried out and the other avoided, depending on the choice made.

If the decision merely determines whether or not a section of the program will be bypassed, the bypass can be shown as a narrow column on the page. Similarly, a program loop can be shown as a narrow column going up the left side of the diagram. Diagonal lines or crosshatches are often used, as they are in [Figure 4-1](#), to indicate that this is the path of a closed loop, rather than a forward path within the program.

OPEN DATA FILE

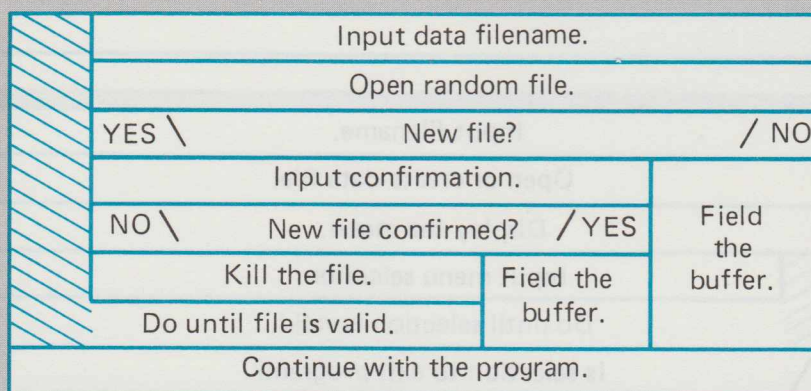


Figure 4-2. The Chapin diagram for opening the PHONEBOOK data file.

```

100 rem startup & open disk file
105 rem
110 poke 53280,14:poke 53281,14
120 printchr$(14);"[BLU][CS] PHONE BOOK and MAILING LIST HANDLER";
130 input"[CD][CD][CD]What is the name of your data file      on disk";f1$
140 open 1,8,15:open 2,8,8,f1$+",1,"+chr$(254):gosub 900
150 r=1:po=1:gosub 800:if e=0 then 200
160 print"[CD]";f1$;" is a new file."
170 input"Is this what you want";a$:if a$="y" or a$="Y" then 200
180 close 2:print#1,"s:"+f1$:print#1,"i":close 1
190 goto 100
195 rem
200 rem define file fields
205 rem
210 ln=1:na=21:mi=36:ad=37:ct=71:st=96:zp=98:hp=103:wp=113:n1=123:n2=156
220 n3=189:n4=222

```

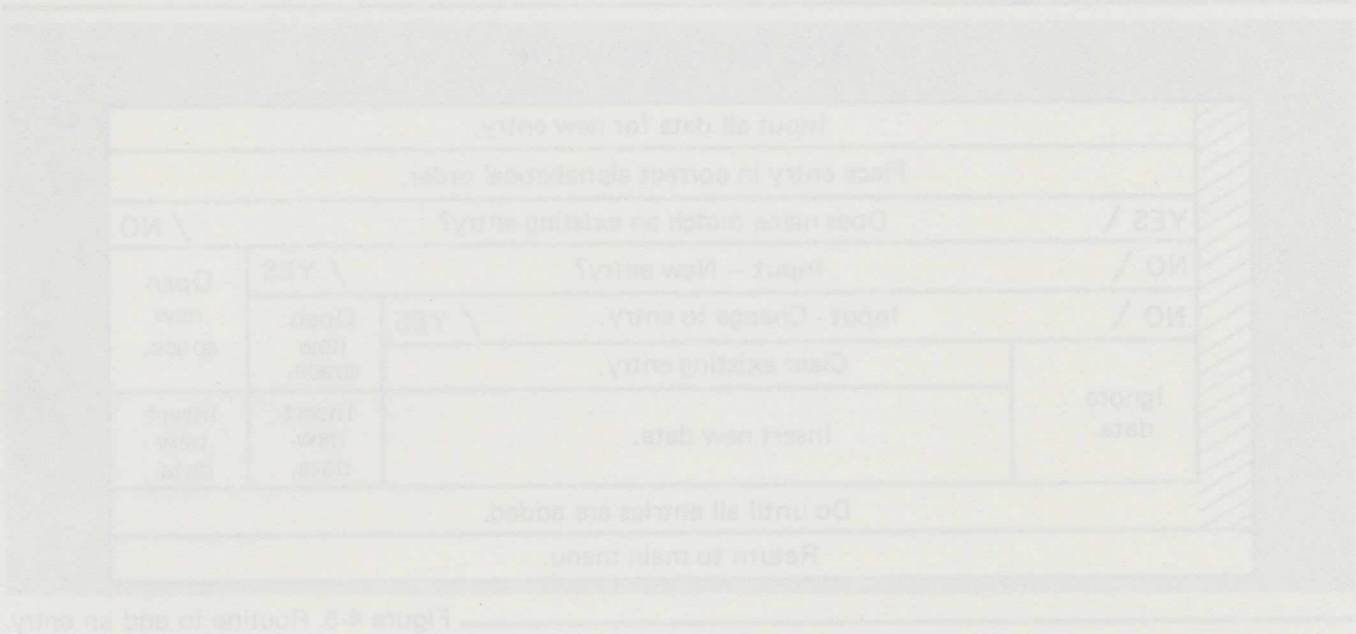
Figure 4-3. Opening the PHONEBOOK data file.

Sector 3: Diagramming an Internal Routine

Chapin diagrams for the internal program modules or routines follow exactly the same pattern. The only difference is that each second-level diagram corresponds to one and only one of the lines in the primary diagram of Figure 4-1. Then third-level diagrams may be drawn as needed, and so on, until every program step is fully detailed.

Figure 4-2 shows the Chapin diagram for opening the data file for PHONEBOOK. Compare this diagram with the program listing in Figure 4-3. Does the program listing correctly reflect the steps given in the Chapin diagram?

Selecting the menu option is a little more complicated, but is still quite straightforward. Figure 4-4 shows the diagram for this sequence. Note that at the top we have not only identified the routine, but



**SELECT MENU OPTION
VALID SELECTION ALREADY VERIFIED
NOT PROGRAM EXIT**

NO \	Menu option 1?			/ YES
NO \	Menu option 2?			/ YES
NO \	Menu option 3?		/ YES	Add entry routine.
NO \	Menu option 4?		/ YES	
Menu option 5 = Print labels routine		Display entry routine.	Edit entry routine.	
Delete entry routine.				
Return to main menu display.				

Figure 4-4. Selecting the menu option.

we also state that we have verified that we have a valid menu selection, and that we will not exit the program with this selection (these tasks were detailed in the main diagram).

The Chapin diagram does not support a “case” type of statement, where multiple choices can be directly handled. As a result, the Chapin diagram in

Figure 4-4 shows a series of tests for each menu option in sequence. In the event of a successful test, the appropriate option routine is called, and control leaves this test sequence.

As before, we took some liberties with the Chapin diagram on the computer display. The correct diagram, shown in Figure 4-4, gets progressively

ADD ENTRY ROUTINE

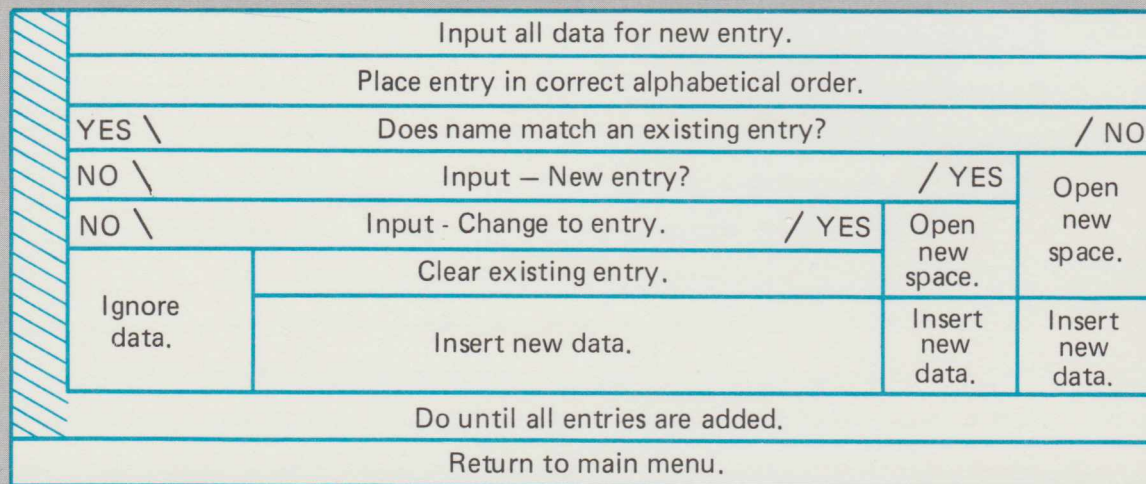


Figure 4-5. Routine to add an entry.

DELETE ENTRY ROUTINE

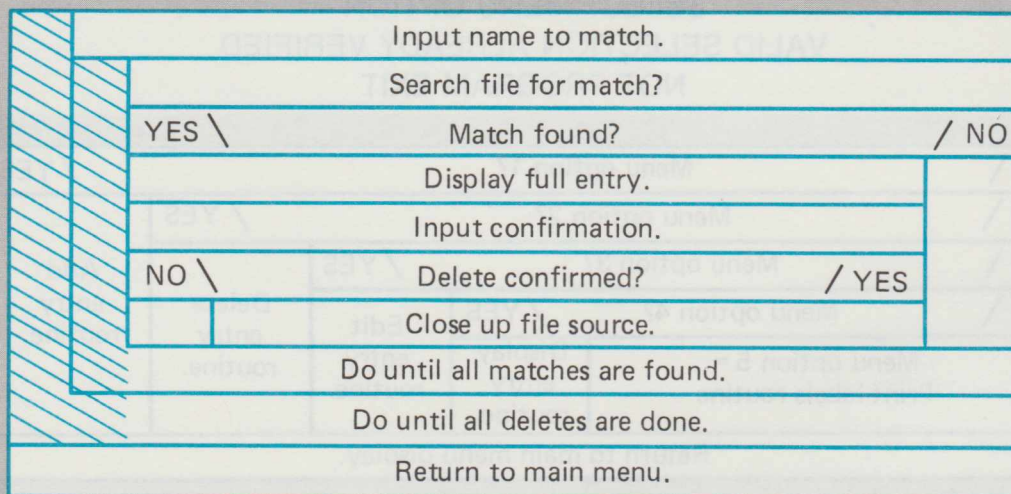


Figure 4-6. Routine to delete an entry.

narrower as additional tests are conducted in sequence. This is typical of Chapin diagrams with this kind of test structure.

The Chapin diagrams for the individual option routines are shown in Figures 4-5 through 4-9. Looking over these diagrams, we can see the requirement of searching through the file for specific names (or other fields) and displaying any match

that may be found. Clearly, this is a likely candidate for a subroutine.

We may also want essentially the same keyboard input subroutine that we defined for the CHECK-BOOK program, at least to add new records to the data file. However, since we have already defined that routine using a flow chart, there is no need to redefine it now.

EDIT ENTRY ROUTINE

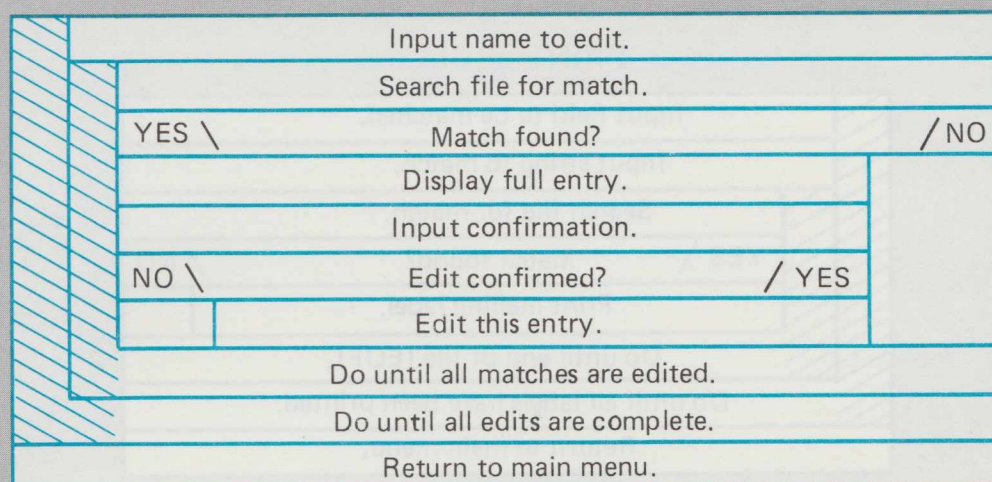


Figure 4-7. Routine to edit an entry.

DISPLAY ENTRY ROUTINE

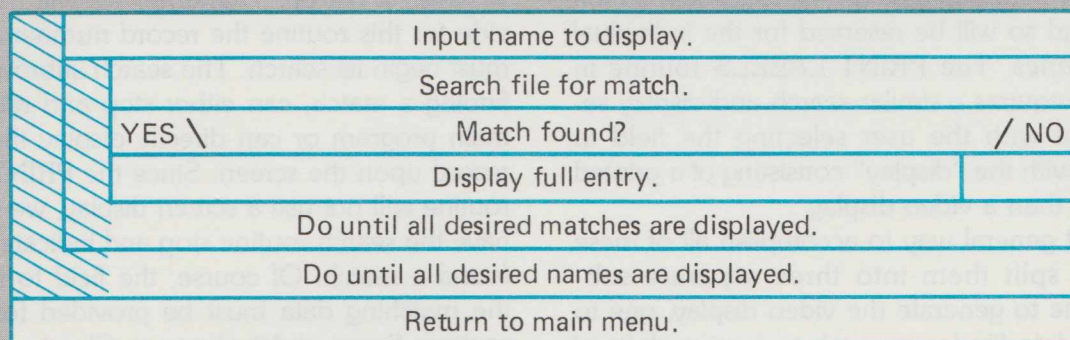


Figure 4-8. Routine to display an entry.

Sector 4: Diagramming Subroutines

As you might expect, diagramming a subroutine is no different than diagramming any other routine within a program. The only requirements are that you identify the subroutine, and indicate somehow that data get passed back and forth between the subroutine and the main program. This is simply the basic information you would need to be able to use the subroutine in the first place.

Before we can diagram the subroutine, however, we will first need to decide exactly what we want it to do, and what will be left to other routines. With this in mind, let's look again at [Figures 4-5 through 4-9](#) and see what we will need.

The first four figures all require that the data file be searched for a matching name, and that the matching entry, if found, be displayed in full. The

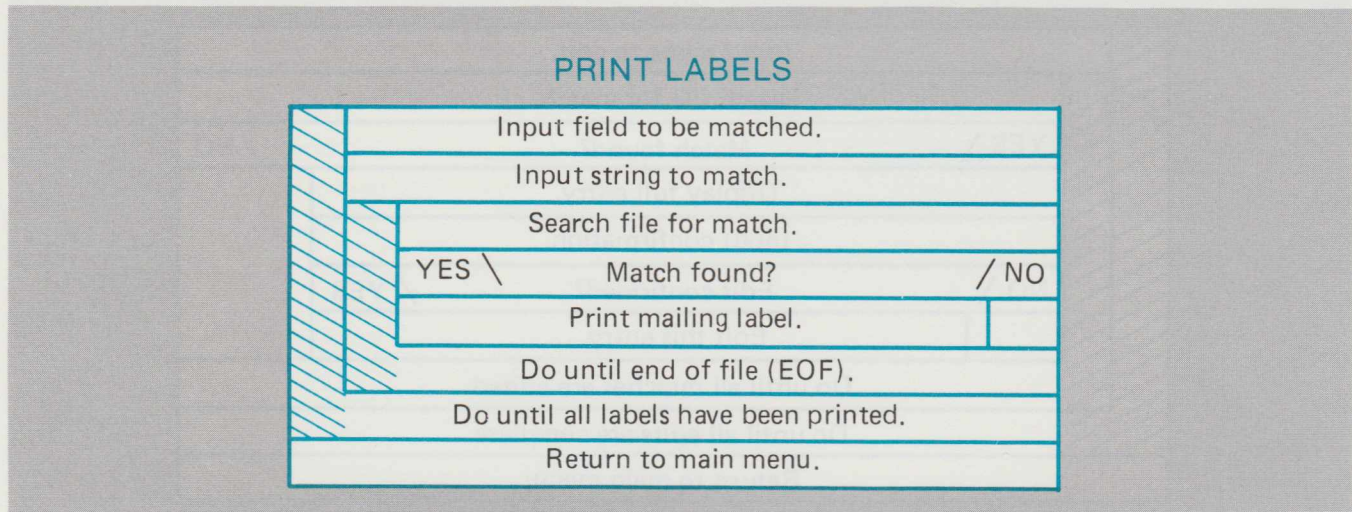


Figure 4-9. Routine to print labels.

action taken then depends on the individual option selected, and so will be reserved for the individual option routines. The PRINT LABELS routine in Figure 4-9 requires a similar search-and-display sequence, but with the user selecting the field to match and with the “display” consisting of a printed label rather than a video display.

The most general way to accomplish all of these tasks is to split them into three separate sub-routines: one to generate the video display, one to search the data file for a match to any predefined field, and one to select the desired field and the specific contents to match. This point at which the search is to start must also be specified for the search routine.

The video display routine was your practice problem from the previous module, so we will work on that later on. For the present, let’s look at the requirements for searching the data file for a particular entry.

Looking again at the Chapin diagrams in Figures 4-5 through 4-9, we find that any search through the data file must first start at the beginning of the file, but then may have to continue from the point

at which it left off. Therefore, we will have to provide for this routine the record number at which it must begin its search. The search subroutine, upon finding a match, can either stop and return to the main program or can directly display the matched record upon the screen. Since the PRINT LABELS routine will not use a screen display, we will simply have the search routine stop and indicate that it has found a match. Of course, the field to match and the matching data must be provided for this subroutine. Figure 4-10 shows a Chapin diagram to accomplish this task.

The subroutine diagrammed in Figure 4-10 will search the currently open data file for the first record it can find that matches the prespecified data. To find multiple matches, it will have to be called any number of times. We will accomplish this from a second subroutine, which will be called by most of the option routines. This second subroutine will request from the user the field to match and the data to look for. It will also call the display subroutine if a match is found, and ask the user to either continue or end the search process. As a final item, it should indicate a failure to find a

SEARCH DATA FILE FOR MATCH

INPUTS:

Starting Record #
Field to Match
Matching Data

OUTPUTS:

Match Found Flag (True/False)
Matching Record #

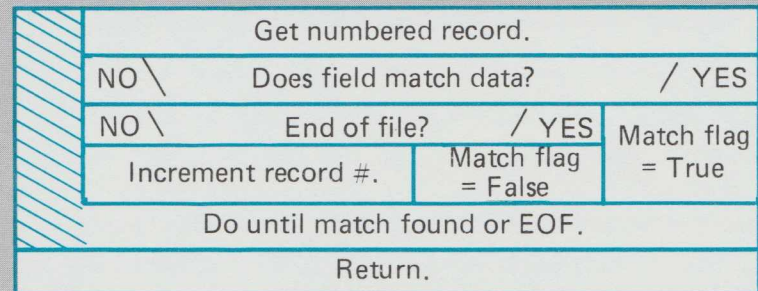


Figure 4-10. The Chapin diagram for the data match routine.

MAIN FILE SEARCH SUBROUTINE

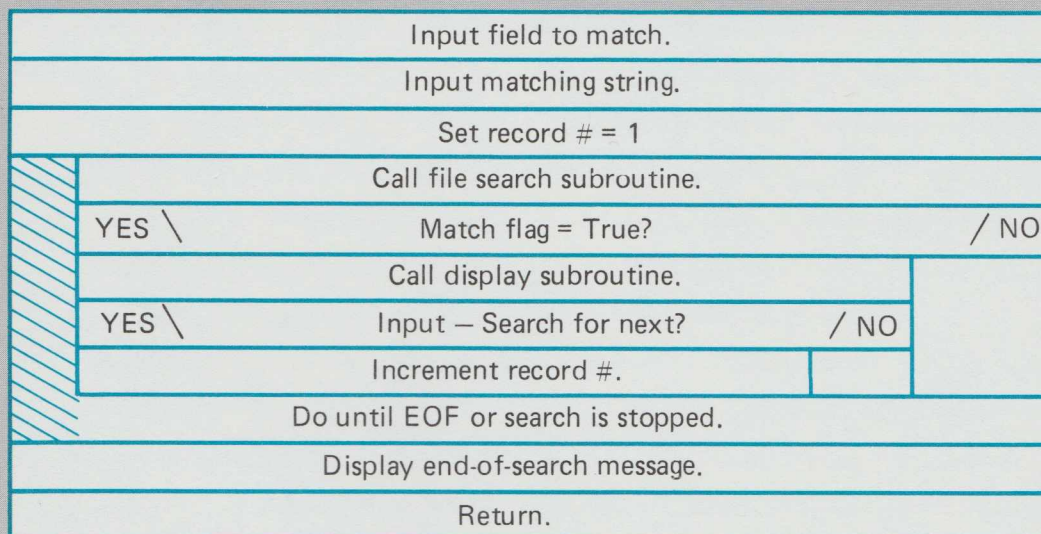


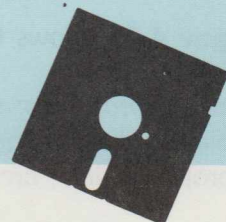
Figure 4-11. The main file search subroutine.

match. Figure 4-11 shows the diagram for such a subroutine.

Additional subroutines, such as the routine to draw the blank video display and another one to fill in the appropriate data, are diagrammed in essen-

tially the same way. When finished, the set of Chapin diagrams forms a complete and detailed description of the program, just as the final set of flow charts does for that kind of diagramming procedure.

NOTES:



TRACK 5

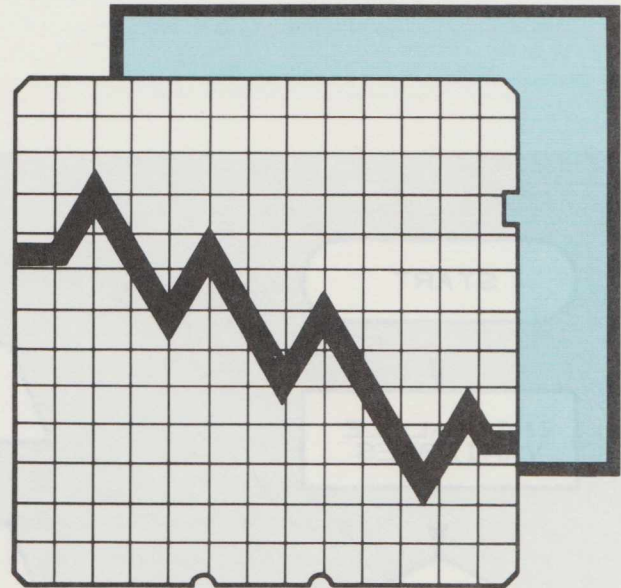


CORRECTIONS AND ADJUSTMENTS

The flow charts and Chapin diagrams presented in the last two tracks were shown in their final form, as if it were only necessary to draw them once, and they would automatically be correct. This is not actually the case. As you think about the requirements of a program while looking over a particular flow chart, you will often recall a step that you left out. Or, looking over several related charts, you will notice that a subroutine could be modified to some extent, and thereby simplify the remaining program considerably. Of course, the same conditions apply to Chapin diagrams or to any other graphic description of the program.

In fact, one of the primary reasons for graphing out the solution procedure to a problem is that this process shows up such gaps and relationships that a simple outline or verbal description could never find. Graphical procedures clarify the relationships between various parts of the solution procedure, and point up various similarities and differences.

The usefulness of the graphical approach, however, cannot guarantee that the first chart or diagram will be complete and fully accurate. In fact, this is very unlikely. Therefore, we must make provisions for updating and adjusting such charts.



If your computer is available at this time, turn it on if necessary and insert your program disk in Drive #8. Then type in the commands:

```
LOAD "TRACK5",8<return>
RUN<return>
```

Return to this point when your computer instructs you to do so.

Sector 1: Correcting and Extending a Flow Chart.

Whenever you draw up a flow chart, you already know that it will most likely have to be extended and corrected, and eventually redrawn. As you gain experience in the programming procedures, you will find yourself thinking automatically of the many small things that are needed to make a program work. However, even then you will find yourself making various adjustments and modifications to your flow chart, so as to improve the operation of the program itself.

The basic rule of flow-charting in this respect is: Don't try to make it perfect the first time! Instead, go ahead and draw your chart as it exists in your

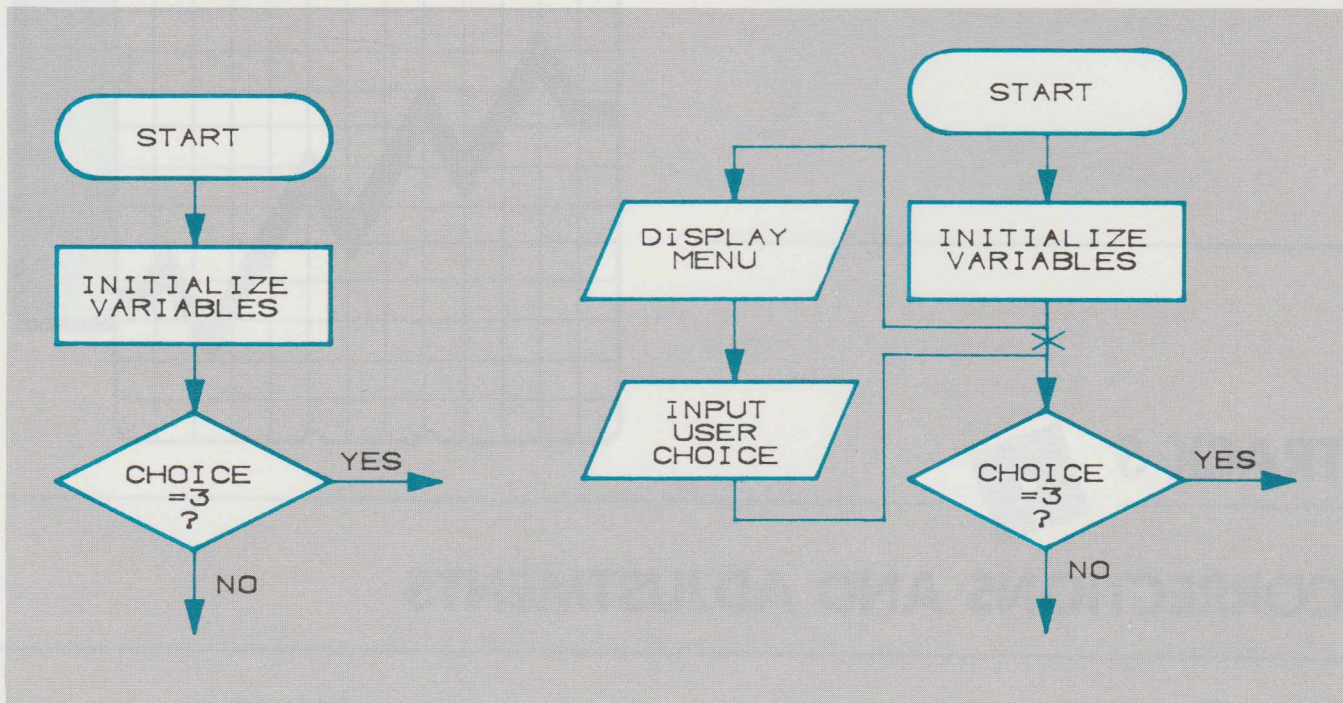


Figure 5-1. Part of a typical flow chart.

Figure 5-2. Inserting the missing steps.

mind. Then you aren't as likely to forget an idea or get lost thinking about it. You can still insert quite a number of modifications to the chart, and then redraw it in its corrected form.

To make the correction process easier, it is a good idea to leave a reasonable amount of space between successive boxes in a given column of the chart, and to leave enough room between columns to insert another column. If this means that you do not have enough room on the page to complete the flow chart, break it in half and put it on two pages. Normally, the first-round flow chart will not contain complete detail, and should fit on one sheet. If it does not, you are probably trying to do too much at one time.

Figure 5-1 shows the first few step boxes in a flow chart. The nature of the program is unimportant, as is the exact purpose of the program. However, note that something has been left out: we have the initialization of system variables (which is always necessary), followed by a decision based on the current value of some variable. The missing

items are the user input which defines the CHOICE specified in the decision box, and a menu display to tell the user what choices are available. Without an input from the user, the variables will all be in their initialized states, and we already know (from the initialization routine itself) what the value of CHOICE will be.

We want to insert the required I/O boxes to denote these factors, but we don't want to redraw the entire flow chart as yet. We will check the remainder of the flow chart for errors or omissions and make all the corrections we can before we redraw the chart. Instead for the moment we mark up the present chart as shown in Figure 5-2.

What we have done here is placed an X across the line from the initialization process box to the decision box. This denotes that the line is broken, and will not be used. Then we draw the missing I/O boxes beside the column, and draw new lines to include the boxes in the program sequence. Of course, we also fill in the boxes with the functions they are intended to perform.

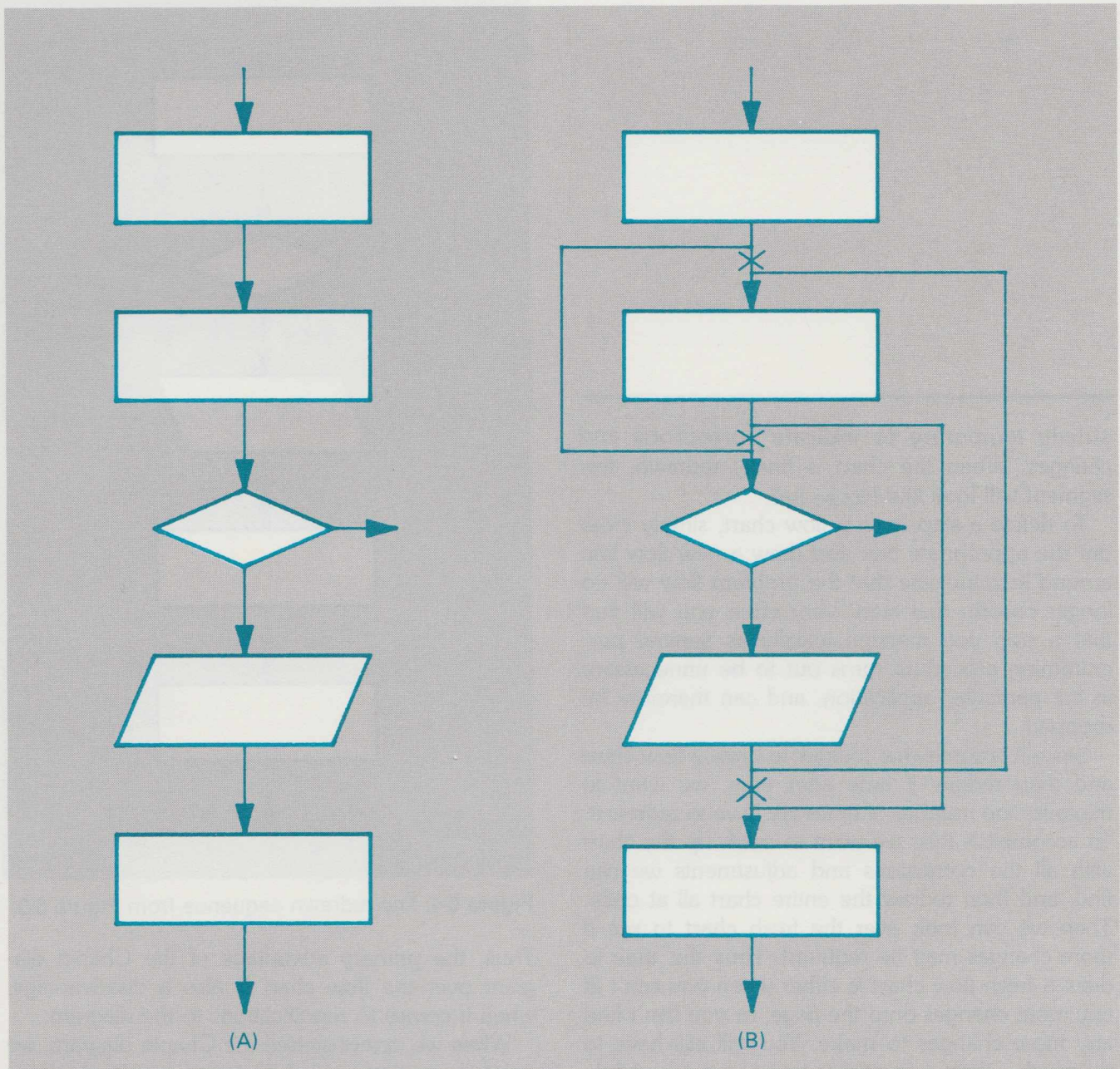


Figure 5-3. Changing the sequence of steps in a flow chart. (A) Before. (B) After.

Another typical problem occurs when you look over your flow chart and decide to change the sequence of the steps. You may originally lay out the chart using the sequence you expect to use, and then find that you need to rearrange the order in which the steps are to be performed. As before, you do not want to actually redraw the flow chart

yet. Therefore, you will once again use X's to break the lines between boxes, and then draw new lines to show the corrected sequence. Figure 5-3 shows the before and after charts for such a change.

In Figure 5-3(B), we have broken several flow-charting rules by having lines go in many different directions, cross each other, etc. However, this is

strictly temporary to indicate corrections and changes. When the chart is finally redrawn, this segment will look like [Figure 5-4](#).

To delete a step from a flow chart, simply cross out the appropriate box and draw a new flow line around it to indicate that the program flow will no longer contain that step. Very often you will find that a step you inserted initially as general programming procedure turns out to be unnecessary in the particular application, and can therefore be dropped.

Since it is somewhat tedious to draw a flow chart and then redraw it time after time, we want to minimize the number of times we have to redraw it. To accomplish this, we want to mark up the chart with all the corrections and adjustments we can find, and then redraw the entire chart all at once. Then we can look over the fresh chart to see if more changes may be required. Thus the time to draw a fresh flow chart is either when you can't fit any more changes onto the page, or you can't find any more changes to make. You will still have to redraw the chart a few times to make it completely correct, but you can at least cut the number of drawn charts to a minimum.

Sector 2: Correcting and Extending Chapin Diagrams

The Chapin diagram is a bit harder to correct than a flow chart, because of its more compact structure.

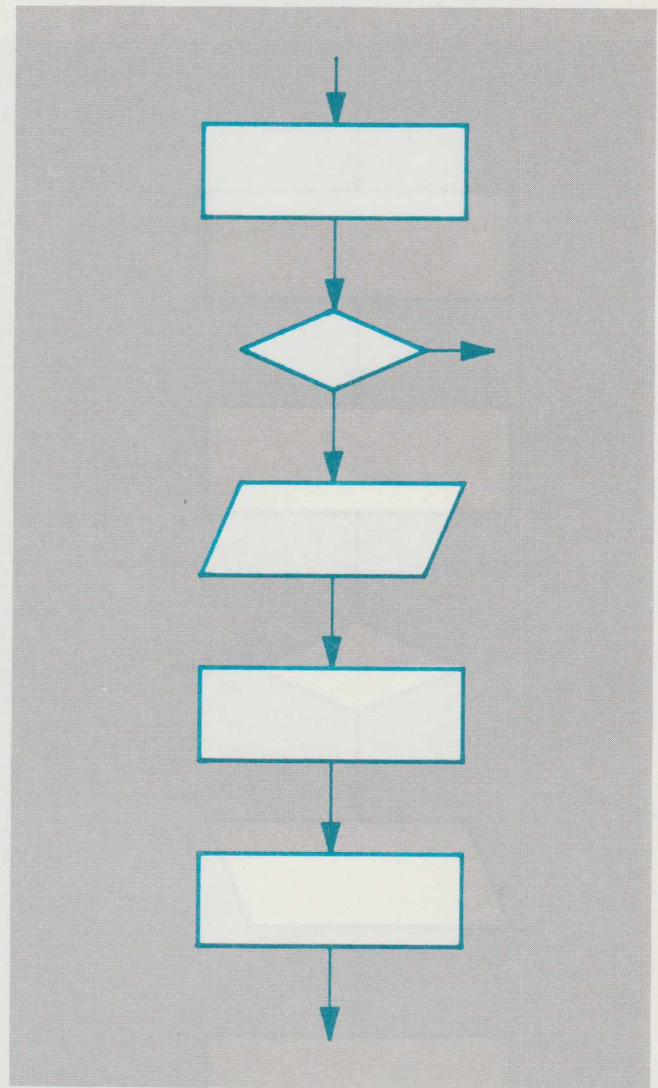


Figure 5-4. The redrawn sequence from [Figure 5-3](#).

Thus, the primary advantage of the Chapin diagram over the flow chart is also a disadvantage, when it comes to modifications to the diagram.

When we demonstrated the Chapin diagram, we pointed out that nested decisions rapidly decrease the working width of the diagram, so it is desirable to start out using the entire width of the paper. At the same time, however, we want to leave some room for corrections and modifications. We can't put these in the middle of the diagram, so we want to leave some margins on either side of the page for such notations. This is especially important for the earlier diagrams, since they are more likely to require corrections.

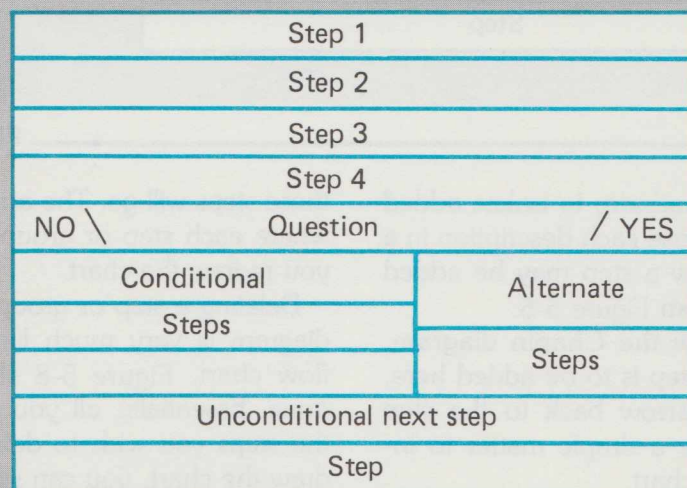


Figure 5-5. A partial Chapin diagram.

Consider the skeleton Chapin diagram shown in [Figure 5-5](#). This shows a part of some program (we don't care what it does for the present). Now suppose that you, the programmer, decide that you want to exchange steps 3 and 4, insert a new step between steps 1 and 2, and delete the ALTER-STATE STEPS that go with the YES answer to the QUESTION.

As with the flow charts we looked at before, you don't want to redraw the Chapin diagram yet, as

you may find other changes you wish to make. How, then, do you indicate these changes while waiting to redraw the chart? You certainly don't want to try to remember them. You will probably forget most changes while looking for other problems.

To add a step to the Chapin diagram, write its description in the margin near the point where it is to be inserted. Then draw an arrow from the new description to the line between the two steps where

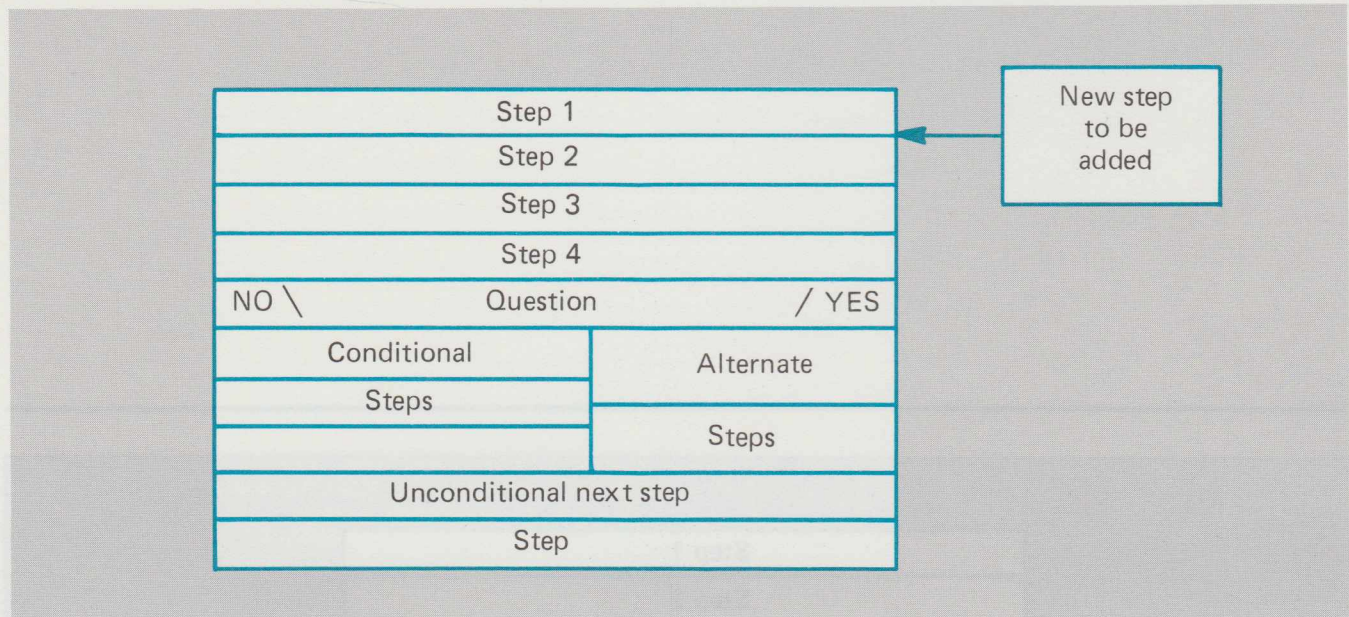


Figure 5-6. Adding a step.

the new step is to go. If necessary to isolate added steps from each other, enclose each description in a box. Figure 5-6 shows how a step may be added between Steps 1 and 2 from Figure 5-5.

Later, when you redraw the Chapin diagram, you can easily see that a step is to be added here, and you can follow the arrow back to the step description. Then it will be a simple matter to include the step in the new chart.

To exchange two steps, use the procedure shown in Figure 5-7. Here we have added the exchange of Steps 3 and 4 to Figure 5-6. Note that we have drawn a brace at one end of Step 3 and another brace at the opposite end of Step 4. These braces show that only one step is to be moved in each direction, up and down. If a group of steps must be moved, then one brace will encompass all of the steps to be moved up, and the other will indicate all of the steps to be moved down.

It is even possible to exchange steps *around* another step which will not be disturbed. This is why we first show braces to indicate what steps are to be moved, and then add arrows to show where

these steps will go. The arrows then indicate clearly where each step or group of steps is to go, when you redraw the chart.

Deleting a step or group of steps from a Chapin diagram is very much like deleting steps from a flow chart. Figure 5-8 shows how this may be done. Essentially, all you need to do is cross out the steps you wish to delete. Then when you redraw the chart, you can simply ignore the crossed-out steps, and close up the diagram to eliminate their spaces.

Try to be neat when crossing out steps in this fashion. It is always tempting to draw very heavy lines and cross out steps with a flourish. However, doing this may lead you to accidentally cross out more than you mean to. Also, wide, heavy lines make it difficult to read what you had written for that step, and you may have to read it later on to confirm that you really meant to delete that step. Always leave open the possibility that you may have made an error, or that you may change your mind and have to reinstall the step you previously deleted.

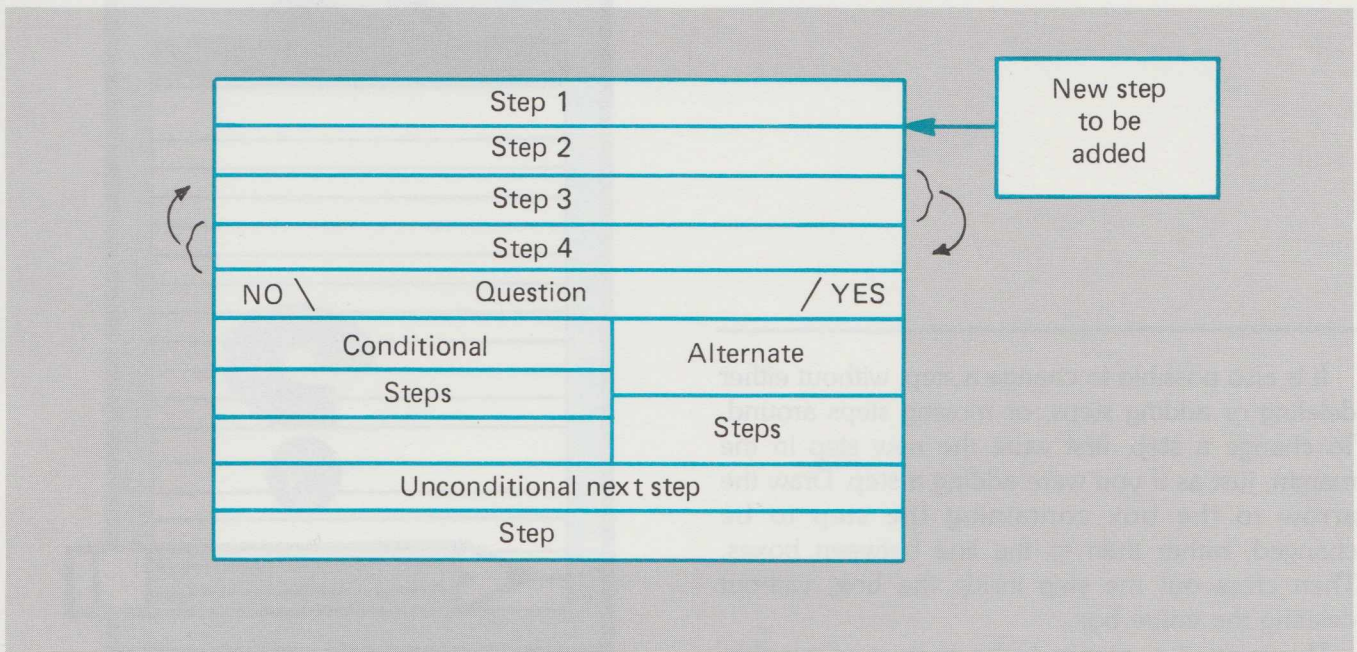


Figure 5-7. Exchanging steps.

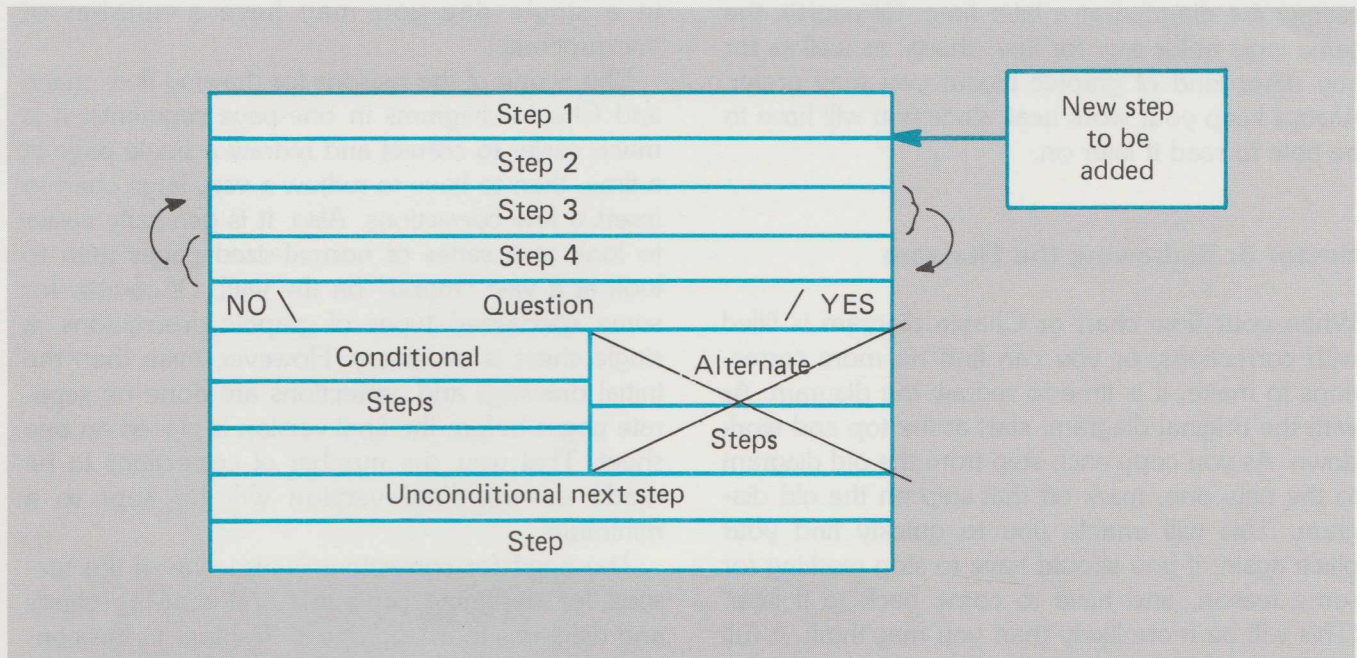


Figure 5-8. Deleting steps.

It is also possible to change a step, without either deleting or adding steps, or moving steps around. To change a step, first write the new step in the margin, just as if you were adding a step. Draw the arrow to the box containing the step to be changed, rather than to the line between boxes. Then cross out the step inside the box, without deleting the entire box.

This is another reason to be as neat as possible when modifying a Chapin diagram: you will always want your changes to be clear to you, when you redraw the diagram at a later time. Of course, the same logic holds true for flow charts, as well as for any other kind of graphic layout you may prefer. *Always keep your work neat, since you will have to be able to read it later on.*

Sector 3: Redrawing the Diagram

When your flow chart or Chapin diagram is filled with corrections, or you can find no more corrections to make, it is time to redraw the diagram. As with the original diagram, start at the top and work down. As you copy each step from the old diagram to the new one, mark off that step on the old diagram. This will enable you to quickly find your place again, if you should have to stop working for some reason, and have to come back to it later. (This will be more likely than you may think. A full set of charts for a large program may take days or even weeks to draw, adjust, redraw, and finalize.



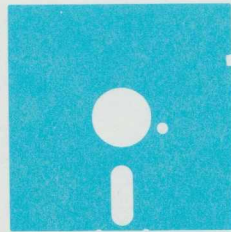
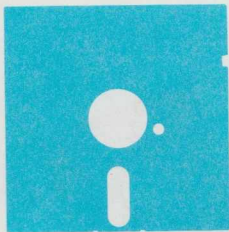
Most programs cannot be done in a day, and even in a single day you may have a number of interruptions.)

This is one of the reasons for drawing flow charts and Chapin diagrams in one-page segments. It is much easier to correct and redraw a single page at a time, than to have to redraw a very large chart to insert a few corrections. Also, it is generally easier to look at a series of normal-sized pages than to look at a vast "mural" on the wall. Of course, for some specialized types of graphic descriptions, a single sheet is necessary. However, even then the initial drawings and corrections are done on separate pages before the final version is placed on one sheet. That way, the number of corrections to be made on the final version will be kept to a minimum.

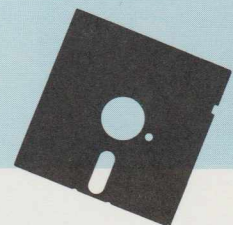
The need for corrections is also one of the reasons for designing programs and drawing charts and diagrams in a "top down" fashion. In this context, "top down" refers to the idea of first diagramming the general procedures and concepts, and

then using separate diagrams or charts to add detail to each step independently. The later, fully detailed charts will be used to actually generate program code, but the earlier charts are still available to show and check the general program structure and overall logical flow. Failure to observe this procedure leads to programs with successive logical steps often widely separated. This kind of “spaghetti code,” as it is called, is extremely difficult to understand, document, or debug later on.

Very often as you are flow-charting or diagramming a program, you will be uncertain as to what sequence of operations you want to use in a particular section of the program. In such a case, simply go to the computer and try it out both ways (or all three ways). There is no reason in the world to guess at this sort of thing, and it is not necessary to stay away from the computer while you are charting your program. What is desirable is that you complete and correct your charts and diagrams before you attempt to code your program. Otherwise, you will find yourself modifying your code too many times, when it isn't really necessary.



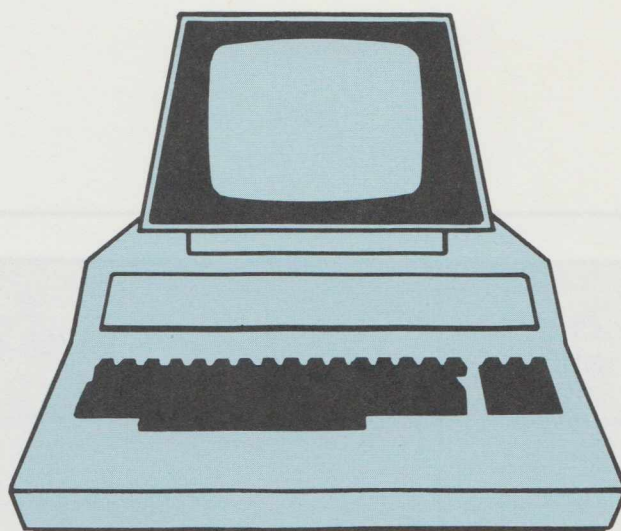
NOTES:



TRACK 6



SOME PRACTICAL EXPERIENCE WITH THE COMPUTER



As you already know, computer programming involves both theory and practice. Furthermore, the two go hand in hand to work together.

In many walks of life, practical computer experience is not presently necessary. However, this is rapidly becoming less true, as our society uses computers in more different ways. Even for those people who need not deal with computers directly, an understanding of computers and computer programming may prove to be helpful at almost any time. Eventually, we can expect that almost everything we do, at least in terms of transactions with other people, will involve computers.

This is why every module in this series includes some practical work with the computer as well as theory. If you understand computer programming and operation, you can be in control of the system as it applies to you. You need not be at the mercy of computer programmers who are not familiar with your requirements and your particular situation.

In the last track of Module 2, the practice problem we assigned was to develop the video display routines for the CHECKBOOK and PHONEBOOK programs. We did not specify precise procedures to follow, although we did suggest that you use the same methods that we employed in our TRACK programs on the Program Disks. We also specified that each routine was to have line numbers in the range 7000-7999. Since the two routines will go with two different programs, there will be no problem with overlapping line numbers.

The reason for the very limited guidance in designing and writing routines was that this way, you would find out for yourself what it is like to think of everything from scratch, and to design and write a routine without first performing the complete organization of it on paper. Of course, an experienced programmer might very well organize this

CHECK NO. (4)	PAYEE (20)		AMOUNT (8)	
DATE (5)	FOR (20) (PURPOSE)		TAX (1)	CHECK FEE (4)
	DATE (5) CANCELLED	STATEMENT DATE (5)	BALANCE (9)	

Figure 6-1. The video display format for CHECKBOOK.

particular routine mentally and then keyboard it directly. However, this kind of experience comes first from organizing such routines on paper, before coding the program.

The program disk enclosed with this module includes possible video display routines for both exercise programs. They are named CHECKVID and PHONEVID. You will need these two programs, your own versions of these display routines, and the main CHECKBOOK and PHONEBOOK programs for Module 2.

Sector 1: The Video Display Routine for the CHECKBOOK Program

In Module 2, we defined the requirements for the blank display for the CHECKBOOK register. Figure 6-1 shows this display, with the names of the fields and the number of characters in each field filled in. These field names will head the overall register dis-

play, but will not be part of this particular routine. All we want to do for the moment is to draw the lines required to form the blank display. To simplify matters, we will assume that the bottom line of one display will not be used as the top line of the next one. Rather, the next blank display will appear one line below the previous one, assuming there is room for it on the screen.

To accomplish this, we must first do two things: we must flow-chart (or otherwise diagram) our routine sequence, and we must determine what characters we can use to draw the lines. The line characters will vary according to the computer and its video character generator. For example, the TRS-80 Model I, Model III, and Model 4 have a coarse graphics capability in which each character position may be specified as a two-by-three block cell, and each block may be turned on or off independently of the others. Other computers have different techniques, which are reflected in the programs on the enclosed disk. The IBM PC and

PC compatible computers have a set of graphics characters, shown in Figure 6-2, which can be accessed using the CHR\$ function. In the Commodore computers, we can use the graphics characters available with the C= and SHIFT keys.

Now we need the flow chart to draw the blank display. This is not the same as the flow chart we developed to *fill in* this display, earlier in this module. However, we will use the exact same starting point, which is the upper left-hand corner of the individual display. We will need to draw 7 strings of characters on the screen, as shown in Figure 6-1. Each string will consist of appropriate graphics characters, possibly mixed with spaces to make room for later data entries. Figure 6-3 shows the basic flow chart for this routine.

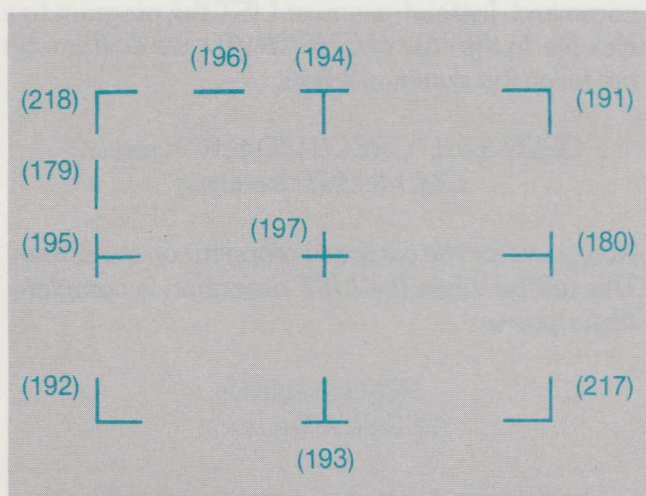


Figure 6-2. The single-line graphics characters available on the IBM PC and PC compatible computers, along with their CHR\$ values.

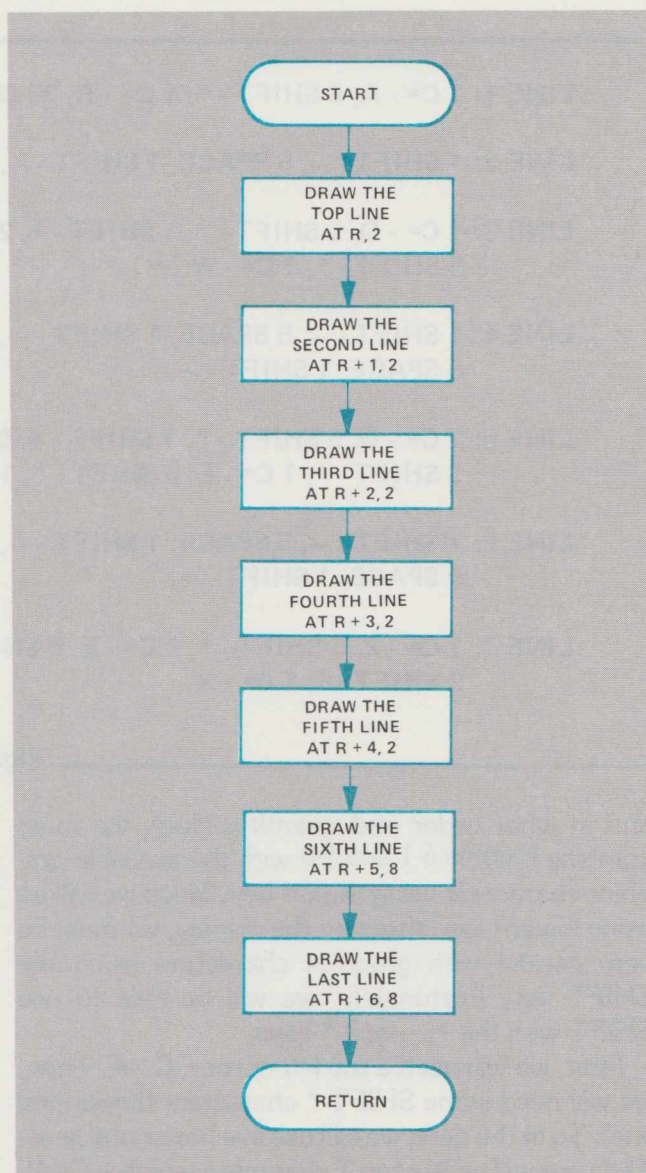


Figure 6-3. Flow chart for the blank video display routine in CHECKBOOK.

The flow chart in Figure 6-3 may seem a bit simplistic and possibly unnecessary, but it is a good idea to draw it anyway, at least at first. It will help to organize your thinking, until you are used to working with computers. Keep in mind that computers are strictly literal; they cannot insert an "obvious" step that you neglected to state explicitly.

Now that we know exactly what we want to do and where we will put each line, we must determine which characters will be used in which lines,


```

LINE 1: 1 C= - A, 5 SHIFT - *, 1 C= - R, 20 SHIFT - *, 1 C= - R, 9 SHIFT - *, 1 C= - S.

LINE 2: 1 SHIFT - —, 5 SPACE, 1 SHIFT - —, 20 SPACE, 1 SHIFT - —, 9 SPACE, 1 SHIFT - —.

LINE 3: 1 C= - Q, 5 SHIFT - *, 1 SHIFT - +, 20 SHIFT - *, 1 SHIFT - +, 3 SHIFT - *, 1 C= - R,
        5 SHIFT - *, 1 C= - W.

LINE 4: 1 SHIFT - —, 5 SPACE, 1 SHIFT - —, 20 SPACE, 1 SHIFT - —, 3 SPACE, 1 SHIFT - —,
        5 SPACE, 1 SHIFT - —.

LINE 5: 1 C= - Q, 5 SHIFT - *, 1 SHIFT - +, 9 SHIFT - *, 1 C= - R, 10 SHIFT - *, 1 SHIFT - +,
        3 SHIFT - *, 1 C= - E, 5 SHIFT - *, 1 C= - W.

LINE 6: 1 SHIFT - —, 5 SPACE, 1 SHIFT - —, 9 SPACE, 1 SHIFT - —, 10 SPACE, 1 SHIFT - —,
        9 SPACE, 1 SHIFT - —.

LINE 7: 1 C= - Z, 5 SHIFT - *, 1 C= - E, 9 SHIFT - *, 1 C= - E, 10 SHIFT - *, 1 C= - E,
        9 SHIFT - *, 1 C= - X.

```

Figure 6-4. The contents of each line in the blank display.

and in what order and quantity. Here, we must combine [Figures 6-1 and 6-3](#) with the available graphics characters using the C= key. Since we will be using lower-case letters on the display, we must be very careful with graphics characters using the SHIFT key. Fortunately, we will be able to use SHIFT with the +, —, and * keys.

First, we will need a top left corner, C=A. Then, we will need some SHIFT-* characters (horizontal line). To fit the date, we will use five horizontal lines. Then, we will need a top T character, which is C=R, followed by 20 more SHIFT-*'s, another C=R, 9 SHIFT-*'s, and the upper right corner, C=S. This is shown as LINE 1 in [Figure 6-4](#).

The remaining lines of the display are also shown in [Figure 6-4](#). Note that no two lines are identical, so we must define each line individually. This is easy to do directly in a PRINT statement.

The resulting program routine is actually quite short, and requires only a few lines to draw the display, with a RETURN statement at the end. You can of course include any desired REMarks as well.

In order to combine this routine with an existing program, it must first be saved to disk in ASCII

format. This is not the usual format used by the Commodore computers, and the Commodore operating system does not really make provision for the technique that we will use, but the capability does exist and we can make use of it.

In order to save a program or any part of a program in ASCII format, we cannot use the SAVE command. Instead, we must LIST the program to a disk file. *In the case of CHECKVID, we do this with the following command lines:*

```

OPEN 8,8,8,"CHECKVID,S,W"<return>
CMD 8:LIST<return>

```

Now, wait for the cursor to reappear on the screen. This will be when the LIST operation is complete. Then type in:

```

SDFG<return>
CLOSE 8<return>

```

The first line can be any illegal command or character string. It produces a syntax error message and terminates output to the disk file CHECKVID. The CMD 8 and LIST command *must* be placed on the

same command line for this technique to work; we will see why shortly.

Now, we can combine this routine with your original CHECKBOOK program as corrected and modified, as follows: *First, insert your program disk containing the latest version of CHECKBOOK into Drive #8. Using whatever program name you assigned to this program, LOAD it into memory, but do not RUN it yet.*

Now, remove the disk from Drive #8 and put it aside in its protective paper sleeve. Insert your program disk from this module, and type in the following list of commands:

```
POKE 812,73:OPEN 8,8,8,"CHECKVID"<return>
POKE 19,8:POKE 781,8:SYS 65478<return>
```

At this point, the computer will start reading the file CHECKVID from disk, and will interpret it just as if you were typing it in from the keyboard. When the computer reaches the end of the disk file, the screen will suddenly show "syntax error," and the computer will be back in keyboard command mode. *Now, type in the following command lines:*

```
POKE 19,0:POKE 812,47:CLOSE 8<return>
```

This restores the computer to normal status, and will allow all of your programs to work normally.

Now, LIST the entire program. Compare the subroutine starting at line 7000 with your version of the same routine. As you do so, remember that the exact method and exact code used to obtain the display do not matter much, as long as the display

itself is correct. This is true of most programs: if the end result is correct, the program is satisfactory. *Demonstrate this routine by typing in the command line:*

```
? "[CS]":R=5:GOSUB 7000<return>
```

How does this display compare with Figure 6-1?

Now, let us see why the computer behaves as it does throughout this procedure. LISTing the file to a disk file requires exactly the same procedure as LISTing it to the printer. The only differences are the device number used and the requirement of specifying a filename for the disk file. At the end of the LIST process, we could issue the command PRINT#8<return> before we CLOSE the file. This is the technically correct procedure. However, we chose to issue an illegal command in order to demonstrate a point: The moment an error occurs, the computer terminates file operations and returns automatically to the keyboard and screen.

When you entered the commands needed to merge CHECKVID with the existing CHECKBOOK program, you basically told the computer to take its input from the disk file instead of from the keyboard. Since this file was saved with line numbers intact, each line remained numbered, and was added to the existing program with the same line numbers as before.

The syntax error caused the computer to terminate input from the file and reset itself for keyboard input. But what caused this error? It was in fact caused by an erroneous direct command in the file. If you LIST a program to the printer, what is the last line you see on paper? It is the "Ready." message. If you try to execute this message as a command, it appears as READ Y., complete with the period. Since Y. is not a valid variable name, BASIC produces a Syntax Error message and falls back into command mode.

	(20)	,	(15)	(1)
			(34)	
	(25)		(2)	(5)
PHONE	(10)		BUSINESS	(10)
NOTES			(33)	
			(33)	
			(33)	
			(33)	

Figure 6-5. The blank display for the PHONEBOOK program.

The same "Ready." message is sent to the disk file when you LIST the program to it. This is why the LIST command must always be given on the same line as the CMD 8 command. If it is not, BASIC will write a Ready. message at the beginning of the disk file, and you will not be able to load it in using this technique.

When you have completed the merging operation, SAVE the complete program back to disk. This will place it on a single disk as a single program.

Sector 2: The Video Display Routine for the PHONEBOOK Program

The video display routine for the PHONEBOOK program is essentially the same as for the CHECKBOOK program. However, there is a major difference between them: CHECKBOOK will put multiple records on a single screen, while PHONEBOOK needs only display one entry, or record, at a time. Therefore, the location of the displayed lines for PHONEBOOK will be fixed, and need not depend on any predefined location.

Figure 6-5 shows the video display format for PHONEBOOK, as we developed it in Module 2. In

this display we will include some descriptive words, spaces, and other characters as well as the lines that will indicate the size of each field in the record. However, due to space limitations on each line, we cannot include the words NAME, ADDRESS, CITY, etc. The top line uses 36 characters for the name, as well as two spaces and a comma. This uses 39 columns on the 40-column display, and to prevent wraparound we do not want to use the last column. Therefore, this line is completely filled.

Similarly, the lines for address and city/state/zip are nearly filled. However, we do have room to include legends for PHONE, BUSINESS, and NOTES, so these are shown in Figure 6-5. We will have to assume the purposes of the top three lines of the display. Fortunately, these are the normal meanings of these lines for this type of application, so we should have no trouble.

Since there are eight lines in this display, we can use eight program lines to produce it, plus a RETURN statement and any desired REMarks. The only graphics character we will need is the horizontal line at the bottom of the character box. This is available as C=@ (the @ key with the C= key held down). These lines will later be replaced by other characters as they are typed in. The initial display, however, will match Figure 6-5.

As with the flow chart for the blank CHECKBOOK display, the Chapin diagram for the PHONEBOOK display is direct and straightforward. Figure 6-6 shows this diagram. Note that the three unlabeled NOTES lines are specified as starting in column 7, rather than in column 1. This will align them vertically with the previous display line which includes the word NOTES and a trailing space.

The file PHONEVID on the enclosed program disk contains our routine for producing this display. *To demonstrate it, first merge this routine with the PHONEBOOK program from Module 2, using the same technique we used for CHECKVID.*

Put name line at 5, 1
Put address line at 6, 1
Put city line at 7, 1
Put "PHONE" line at 8, 1
Put "NOTES" line at 9, 1
Put blank line at 10, 7
Put blank line at 11, 7
Put blank line at 12, 7
Return

Figure 6-6. The Chapin diagram for the blank PHONEBOOK display.

SAVE the combined program, and then type in the command:

`GOSUB 7000<return>`

How does this display compare with Figure 6-5?

Compare our routine with the one you wrote as a practice problem for Module 2. Do they produce the same or equivalent displays, even though the actual code may be slightly different? If so, your routine is perfectly satisfactory for the PHONEBOOK program.

Sector 3: The Use of GET for Nonstandard Keyboard Input

You are already familiar with the INPUT command as a means of requesting user input for a program to use while it is running. This command is highly useful and is entirely satisfactory in many applications. However, the INPUT command does have a few fixed characteristics that are not compatible with a structured display such as we are using for CHECKBOOK and PHONEBOOK. For one thing, INPUT will accept any desired number of characters before you press the <return> key, and we will want to limit our entries to a fixed number of characters. Also, when you press the <return> key, the cursor moves to the start of the next line, just as it does when you are typing in program lines. In many cases, this action also erases the entire next line, and in other cases it erases the remainder of the line on which the entry was made. Clearly, none of this will be acceptable for our purposes. We need an input routine that will accept only the characters in

the range we want, but will still let us back up and correct errors, and, at the same time, treat the <return> key merely as a signal that the input string is complete. Furthermore, our routine must ignore characters beyond the length limit we set for it. Let's take a look at the basic program requirements for such a routine in BASIC.

NOTE: For this sector, you will need your computer, operating in BASIC. If your computer is not available at this time, we suggest that you repeat this sector later, when your computer is available to you again.

Type in the command:

NEW <return>

to clear any program that may currently be in memory. Now you are ready to type in your new routine.



We will begin by typing in a short driver routine, which will select various positions on the screen and various input string lengths, and will then call our keyboard subroutine. When our test routine is done, we should be able to enter any data anywhere on the screen without disturbing any other part of the display.

To begin, type in the following program lines:

```

100 PRINT "[CS]"
110 D$ = "[CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD][CD]"
120 R$ = "[CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR][CR]"
130 B$ = "[CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL]"
140 P$ = "....."
150 R = INT(RND(1) * 20)
160 C = INT(RND(1) * 20)
170 L = INT(RND(1) * 15) + 2
180 PRINT "[HC]";LEFT$(D$,R);LEFT$(R$,C);
    Type up to";L;"characters: ";
190 PRINT LEFT$(P$,L);LEFT$(B$,L);
200 GET K$:IF K$ = "" THEN 200
210 GOTO 150

```

In this program, the initial string definitions are for cursor motion and (for P\$) a count indicator. In lines 150-170, the program randomly selects a screen location and string length. Line 180 positions the cursor and specifies the number of characters, and line 190 prints the periods on the screen to match. Line 190 also backs up the cursor to the beginning of the line of periods, although the cursor itself

remains invisible. This is done without actually erasing the periods.

Line 200 provides a manual delay, so that you can control the rate at which this routine writes to the screen. Later, we will replace this line with a GOSUB statement to call the real keyboard input routine that we will develop. Then, line 210 tells BASIC to go back to line 150 and start over again from that point.

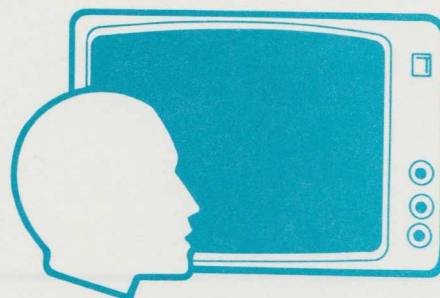
Now that you have entered this routine into your computer, test it by typing in the command:

RUN <return>

Now observe the display. Does the number of periods displayed match the number of characters that you are told to type? Press the space bar (or almost any other key) and see what happens. Repeat this action several times. In each case, is the correct number of periods displayed? What happens if the new display line overwrites part or all of a previously displayed line?

You should have noticed that the position of the new displayed line was indeed random, within the specified limits. If a line landed over part or all of a previous line, the new line simply overwrote the old one. The cursor was invisible. You also found that if the line overflowed the right edge of the screen, it wrapped around to the left edge on the next line. We will have to be careful of this in the real routine.

Now that we know the driver program is working (if you wish, you can reduce the "*" constants in lines 160 and 170 to eliminate the wraparound



effect), we can start coding the keyboard routine itself. We will assign line numbers in the 8000 range to this subroutine, which will later be adapted to both PHONEBOOK and CHECKBOOK. We charted the routine earlier in this module.

Begin with a REMark line to identify what we are trying to do:

8000 REM KEYBOARD INPUT ROUTINE FOR LIMITED CHARACTER INPUT.

Next, we need to initialize the input string. Let's call it KB\$, for KeyBoard input string:

8010 KB\$ = ""

The GET command we used in Line 200 looks briefly at the keyboard to see if a keypress has actually been registered. If so, it returns that character, but if not it returns a null string (""). It does not echo anything to the screen, which is exactly what we want. *Therefore, we will use it to check for a character:*

8020 GET A\$:IF A\$ = "" THEN 8020

Next, we must distinguish between normal characters and control characters. Any character with a ASCII value less than 32 is a control character. This includes cursor keys, <return>, RUN/STOP, INST/DEL, and other such keys. We will handle the two kinds of characters separately.

Therefore, our next line is:

```
8030 IF A$ < " " THEN 8100
```

Of course, we will have to fill in the control character handling routine later, but at least we now know that we won't have to worry about these characters getting into KB\$.

Before we can add a character to KB\$, we must make sure that the string is not already at maximum length. *We can do this with an IF statement:*

```
8040 IF LEN(KB$) = L THEN 8020
```

Now we need only add the new character to the display and to KB\$:

```
8050 KB$ = KB$ + A$
8060 PRINT A$;
```

The main input routine is now complete, and we can look for another character:

```
8070 GOTO 8020
```

With the main routine in place, we must put something in at 8100, or we will get an error message as soon as we press the <return> key. *Therefore, let's allow the routine to recognize the <return> key alone, at least for now:*

```
8100 IF A$ = CHR$(13) THEN RETURN
8110 GOTO 8020
```



Before we can test this program, we must modify the driver program to call it as a subroutine. *To do this, change Line 200 to:*

```
200 GOSUB 8000
```

When you have typed in and checked all of these program lines, RUN the program again and see what happens. Does the program correctly accept only the specified number of characters? Does it recognize the <return> key? Does it recognize DEL or HOME? What does the program do if you type in too many characters? What does it do for cursor motion keys? How about shifted keys?

In Commodore BASIC, the SHIFT key is recognized by adding 128 to the other key that is pressed. Thus, [CD] is recognized as CHR\$(17), and [CU] becomes CHR\$(145). Similarly, [CR] = CHR\$(29) while [CL] = CHR\$(157).

You can determine for yourself the effect of the SHIFT, C=, and CTRL keys on each of the other keys by typing in the following three lines:

```
10 GET A$:IF A$ = "" THEN 10
20 PRINT "[CS]";ASC(A$);
30 GOTO 10
```


Try this now and observe for yourself the effect of these special keys. Can you think of a way to modify Line 8030 to accept these shifted keys as control characters without actually changing the character?

Actually, there are several possibilities here. However, if we examine the codes in binary, we find that the unshifted keys return codes in the range 00000000 through 01111111. That is, the most significant bit is always 0. By holding down the SHIFT key, you set the most significant bit to 1, giving binary numbers 10000000 through 11111111, or decimal numbers 128 through 255. For some graphics characters, the numerical difference is not exactly 128, but the MSB is still set for the SHIFT key. Therefore, we can use a logical AND to mask out the high bit, and test the rest of the code to see if it is a control character. *We can do this by changing Line 8030 to:*

```
8030 IF ASC(A$) AND 127 < 32 THEN 8100
```

The ASC function returns the ASCII code value for a single character. The AND function is normally used for combining two tests in an IF statement, but it will work quite well, provided both values can be expressed as signed 16-bit binary integers. This means they must be in the decimal range -32,768 to +32,767. This is the same range that is required for integers in BASIC. Since the ASC function is limited to single characters, it will always return a number between 0 and 255, so there will be no problem in this application.

With this change, our program will correctly ignore all cursor motion keys, CLR/HOME, and

INST/DEL. It will accept letters and graphics characters, whether produced with the SHIFT or C= keys, and it will act on the 'return' key. It will also ignore any characters that exceed the predefined string length. However, we have no way to make any corrections. We need to add the DEL key to our list of recognized control codes, and add instructions to handle it.

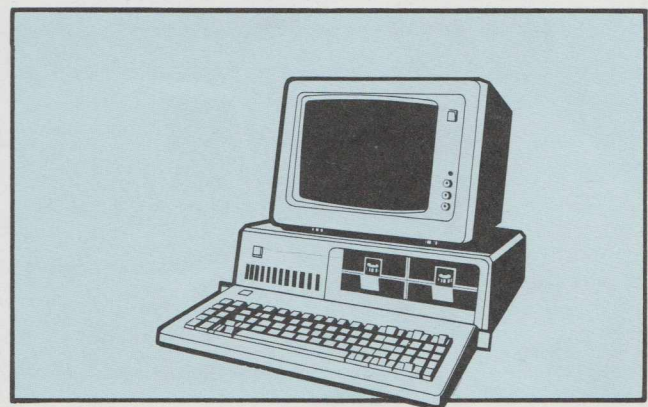
First, change Line 8110 to read:

```
8110 IF A$ <> CHR$(20) THEN 8020
```

CHR\$(20) is the code returned by DEL, so the routine will now recognize this key. *Next, we check to see if KB\$ has zero length, since we don't want to backspace past the start of the string:*

```
8120 IF LEN(KB$) = 0 THEN 8020
```

Assuming that KB\$ has a non-zero length, we must remove the last character from it. We must also erase the character from the display and restore the "." character that was originally there. This requires two different programmed actions.



To clear the character from the screen, we will back up the cursor, write a "." character, and then back up the cursor again. To shorten KB\$, we will use both the LEFT\$ and LEN functions. To implement both actions, type in the following lines:

```
8130 PRINT "[CL].[CL]";
8140 KB$ = LEFT$(KB$,LEN(KB$)-1)
8150 GOTO 8020
```

Now that we have modified the program to recognize DEL, RUN it again to check its operation. Does the DEL key operate correctly? Does the program still recognize normal characters and the <return> key? Does it correctly ignore DEL if you have no characters in the string?

According to our flowchart, we must recognize one more control character. This will be a code that will exit the subroutine without returning a normal

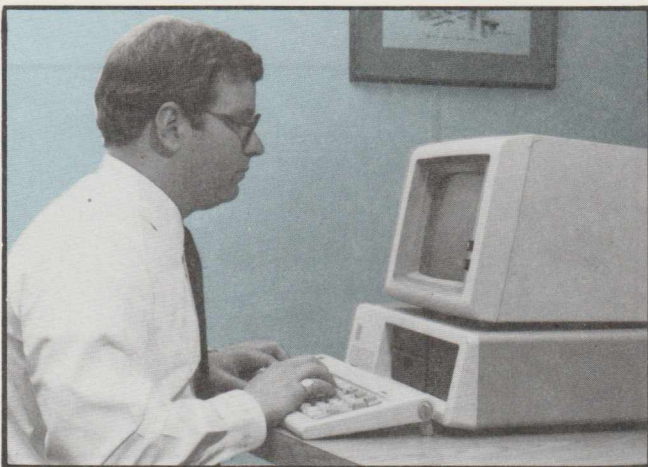
string. We will use this code to "escape" from the subroutine instead of using the normal exit. We have no special ESCAPE key on the C-64, but we have several possibilities. For example, we could use SHIFT-<return>, which normally tells BASIC to ignore this line and go on to the next screen line. Or, we could use the CTRL key and any letter or our choice, such as CTRL-X or CTRL-Q. For compatibility of meaning, we will use SHIFT-<return> here. However, we will use CHR\$(27), which is the normal ESCAPE code, to denote this within the program. To add this capability, type in the following lines:

```
8110 IF A$ <> CHR$(20) THEN 8160
8160 IF A$ <> CHR$(141) THEN 8020
8170 PRINT LEFT$(B$,LEN(KB$));"##";
8180 KB$ = CHR$(27)
8190 RETURN
```

When you have added these lines, test the program once again to make sure that it works correctly in all respects, and that its performance is in accordance with the specifications and flow chart that we drew up earlier in this module. When you are satisfied that this keyboard routine works correctly, use the technique we described earlier to LIST lines 8000 — 8190 to a disk file. You will be merging this routine with both CHECKBOOK and PHONEBOOK as part of your practice project in the next sector.

Sector 4: A Practice Project for Your Next Module

The keyboard entry routine that you typed into your computer in Sector 3 is essentially the keyboard input routine that you will need for both the PHONEBOOK and CHECKBOOK programs.



However, in these programs you will not have periods denoting the number of characters permitted in the input string. Instead, CHECKBOOK will use blanks (spaces), and PHONEBOOK will use the C=-@ character as a straight line.

If you have not already merged the keyboard routine with your latest CHECKBOOK program, do so now using the file procedure specified earlier in this module. Then, make whatever modifications are required to use spaces rather than periods. Test your program to make sure it works correctly.

When you have completed your modifications, SAVE your CHECKBOOK program back to disk. Then, LOAD your PHONEBOOK program and merge your keyboard routine with it. Modify it as necessary to work with the C=-@ character instead of periods or spaces. Then, SAVE your updated PHONEBOOK program back to disk.

When you have completed the modifications to the CHECKBOOK program, SAVE it back to disk. Now LOAD the PHONEBOOK program and MERGE your keyboard input subroutine into this program as well. Modify it to work with the underscore character instead of periods or spaces, and if possible add the BELL character modification described above. When you have completed these modifications, SAVE your PHONEBOOK program again.

When you have completed the modifications to both of these programs, go back to the flow charts and Chapin diagrams we provided for the various internal routines for CHECKBOOK and PHONEBOOK. Choose either program (or both, if you feel daring!) and try coding some of these routines into

your program. Remember to use the range of line numbers already allocated for each routine.

If you are unsure of the proper syntax to use for a given instruction, consult your Operator's Guide or Manual. If you aren't entirely sure of what will happen if you do something, try it and see! You can't hurt the computer even if you give it an impossible instruction; the worst thing that can happen is that you will get a "Syntax Error in Line xxxx". Of course, when you try something involving disk files, make sure that your disk doesn't have something valuable on it that might possibly be erased. If necessary, initialize a new, blank disk in accordance with the instructions in your Operator's Guide and then use it as your practice disk. This is always a good idea when you are trying out unfamiliar file-handling instructions. Then, if something goes wrong, no harm will be done. For the same reason, we suggest that you work with a backup of your program disk, rather than with the original.

Work as much or as little as you please with these two programs. The more time you can spend with them, the more you will learn about your computer and its operation. However, if you don't have the time to spend, or if you find yourself getting lost or frustrated, don't worry! Just put the problem aside for the present. Your program disk in Module 4 will contain complete, working programs for both CHECKBOOK and PHONEBOOK. In addition, we will go into the coding procedures for high-level languages such as BASIC in detail in your next module. We will look for you then!

